13th International Workshop on Worst-Case Execution Time Analysis

WCET'13, July 9, 2013, Paris, France

Edited by

Claire Maiza

Editor

Claire Maiza
Grenoble INP, Verimag
Grenoble, France
claire.maiza@imag.fr

ACM Classification 1998
B.8.2 Performance Analysis and Design Aids, C.3 Real-time and embedded systems, D.2.4 Software/Program Verification

ISBN 978-3-939897-54-5

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/978-3-939897-54-5.

Publication date
July, 2013

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

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

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.WCET.2013.i

OASIcs – OpenAccess Series in Informatics

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

Editorial Board

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

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Evaluation of resource arbitration methods for multi-core real-time systems
Timon Kelter, Tim Harde, Peter Marwedel and Heiko Falk 1

Automatic WCET Analysis of Real-Time Parallel Applications
Haluk Ozaktas, Christine Rochange, and Pascal Sainrat 11

Integrated Worst-Case Execution Time Estimation of Multicore Applications
Dumitru Potop-Butucaru and Isabelle Puaut 21

Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective
Mihail Asavoae, Claire Maiza, and Pascal Raymond 32

Multi-architecture Value Analysis for Machine Code
Hugues Cassé, Florian Birée, and Pascal Sainrat 42

The Auspicious Couple: Symbolic Execution and WCET Analysis
Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr 53

Upper-bounding Program Execution Time with Extreme Value Theory
Francisco J. Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella ... 64

PRADA: Predictable Allocations by Deferred Actions
Florian Haupenthal and Jörg Herter 77

Static analysis of WCET in a satellite software subsystem
Jorge Garrido, Juan Zamorano, and Juan A. de la Puente 87

Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources
Leonidas Kosmidis, Tullio Vardanega, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla 97

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Message from the workshop chair

On July 9, 2013, the 13th International workshop on Worst-Case Execution Time Analysis (WCET 2013, http://wcet2013.imag.fr) will be held in Paris, France. The workshop is being organised as a satellite workshop of the 25th Euromicro conference on Real-Time Systems (ECRTS'13, http://ecrts13.ecrts.org). I am therefore grateful to the ECRTS'13 general chair, Laurent Georges, his local team, and the Real-Time Technical Committee Chair of the Euromicro, Gerhard Föhler, for their work.

The papers that will be presented at the workshop have been selected based on peer reviews by program committee members and external reviewers, all experts in the field. 10 submissions out of 17 were finally selected for presentation. This document contains the presented papers.

I am happy to thank the authors, the program committee including external reviewers, the WCET workshop steering committee for assembling the components of a very successful workshop. The workshop organizers are also deeply grateful to the EU COST Action IC1202: Timing Analysis On Code-Level (TACLe) for financial support. Special thanks to Heiko Falk and Mihail Asavoae for their help.

Claire Maiza

Committees

Program Committee

Guillem Bernat
Rapita Systems, UK

Hugues Cassé
IRIT – Université de Toulouse, France

Francisco J Cazorla
Barcelona Supercomputing Center, Spain

Heiko Falk
Ulm University, Germany

Kevin Hammond
University of St Andrews, UK

Damien Hardy
IRISA, France

Chris Healy
Furman University, USA

Niklas Holsti
Tidorum Lltd, Finland

Björn Lisper
Mälardalen University, Sweden

Tulika Mitra
National University of Singapore, Singapore

Stefan Petters
CISTER/IPP Hurray, Porto, Portugal

Peter Puschner
Technische Universität Wien, Austria

Jan Reineke
Saarland University, Germany

Christine Rochange
IRIT – Université de Toulouse, France

Tullio Vardanega
University of Padua, Italia

Steering Committee

Guillem Bernat
Rapita Systems Ltd., UK

Jan Gustafsson
Mälardalen University, Sweden

Isabelle Puaut
University of Rennes 1 / IRISA, France

Peter Puschner
Vienna University of Technology, Austria

External Reviewers

Mihail Asavoae
Grenoble University / Verimag, France

Marc Boyer
Onera, France

Bekim Cilku
Vienna University of Technology, Austria

Andreas Gustavsson
Mälardalen University, Sweden

Benedikt Huber
Vienna University of Technology, Austria

Daniel Prokesch
Vienna University of Technology, Austria

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza

OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Evaluation of resource arbitration methods for multi-core real-time systems*

Timon Kelter, Tim Harde, Peter Marwedel[1] and Heiko Falk[2]

1 **Department of Computer Science, TU Dortmund**
Otto-Hahn-Straße 16, D-44227 Dortmund
`{timon.kelter,tim.harde,peter.marwedel}@tu-dortmund.de`

2 **Institute of Embedded Systems/Real-Time Systems, Ulm University**
James-Franck-Ring, D–89081 Ulm
`heiko.falk@uni-ulm.de`

Abstract

Multi-core systems have become prevalent in the last years, because of their favorable properties in terms of energy consumption, computing power and design complexity. First attempts have been made to devise WCET analyses for multi-core processors, which have to deal with the problem that the cores may experience interferences during accesses to shared resources. To limit these interferences, the vast amount of previous work is proposing a strict *TDMA (time division multiple access)* schedule for arbitrating shared resources. Though this type of arbitration yields a high predictability, this advantage is paid for with a poor resource utilization. In this work, we compare different arbitration methods with respect to their predictability and average case performance. We show how known WCET analysis techniques can be extended to work with the presented arbitration strategies and perform an evaluation of the resulting ACETs and WCETs on an extensive set of realworld benchmarks. Results show that there are cases when TDMA is not the best strategy, especially when predictability and performance are equally important.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids, D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, multi-core, arbitration, shared resources

Digital Object Identifier 10.4230/OASIcs.WCET.2013.1

1 Introduction

In the last years, many proposals on how to compute safe WCET values for programs running on multi-core systems have been made, but analyses that scale well and are precise at the same time are much harder in the multi-core case than in the single-core one. The central property, that must be accounted for, is that cores may access shared resources and these accesses will have to be arbitrated at some point. Among most of the analysis techniques that have been introduced, it is a common denominator that TDMA should be used for arbitrating the shared resources, since it allows an easy derivation of worst-case bounds for the duration of accesses. In this paper, we compare previously published arbitration methods experimentally both in terms of average and worst-case performance. For the experiments we use a static analyzer and a cycle-true system simulator on a multi-core ARM-platform with a configurable shared bus, which is arbitrated among the cores. We also examine a

* This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-1 and EU COST Action IC1202: Timing Analysis On Code-Level (TACLe).

© Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

more flexible variant of TDMA, called *Priority Division*, and show its effects on ACET and WCET. In summary, the contributions are:

- Generalization of previously published WCET analysis techniques to the mentioned arbitration methods
- Experimental evaluation of average-case and worst-case properties of different arbitration methods on a realistic multi-core platform

The rest of the paper is organized as follows: In Section 2 we will present related work, and Section 3 introduces our system model used in the analyses. In Section 4 the overall analysis framework as well as the changes that are needed to incorporate the different arbitration methods are presented. Section 5 provides the experimental evaluation of the approaches. Finally, we provide a summary of our results and give directions for future work in Section 6.

2 Related Work

The initial scenario covered by WCET analyses was the case of a single program being run uninterruptedly on a single core. This is a well-understood problem for which structured analyses were devised [16]. For the WCET estimation to be safe the analyzer must usually operate on a binary of the analyzed program, since only then locations of code and data are fixed, which affects e.g. cache performance. The problem is separated into control-flow reconstruction, value analysis, microarchitectural analysis and path analysis. *Control-flow reconstruction* can be tackled by a combination of heuristics and data-flow analysis, whereas *value* and *microarchitectural analysis* are usually done as a pure data-flow analysis with the domain of register / memory values or abstract machine states, respectively. In these analyses it may be possible to throw away abstract states that are not inducing a local worst-case, but this is only possible if the architecture under examination is *timing-anomaly-free* [13]. For the last step in this WCET analysis pipeline, the *path analysis*, an integer linear program is formulated and solved, which models all paths through the program together with user-defined flow restrictions, which are needed to bound loops and recursions in the program.

In this work, we also conduct the WCET analysis in the presented way, where the modeling of accesses to shared resources is integrated into the microarchitectural analysis stage with the help of an existing approach for analyzing TDMA offsets [5, 1].

The Priority Division (PD) arbitration policy was first introduced in [14]. The authors provide experimental results of an FPGA-based system which uses PD to arbitrate the shared memory bus, and they state that PD is well-suited for WCET analysis, but this is not investigated further. In this paper we show how to integrate PD into existing WCET analysis frameworks.

The second major category of timing-analysis tools models the multicore system as a set of timed automata and performs model-checking of timing predicates to find the WCET [3]. The drawback of this approach, as discussed in [15], is the possible lack of scalability since the generated models quickly become intractably big. To overcome this, mixtures of abstract interpretation and model-checking have been proposed [8].

Finally, the last approach to deriving multicore WCET values it to derive arrival curves which bound events on the shared resources and compute worst-case access delays from those curves [11]. This has the advantage that the timing behavior of sequences of accesses can be analyzed more easily, therefore these approaches are also called “cumulative” in [2]. On the other hand, the abstraction towards access curves alone already loses some precision.

There have been numerous discussions of arbitration policies for real-time systems. The *PROMPT principles* for predictable multicore architectures [2] advocate "deterministic" arbitration with TDMA as one example. The comparison in [12] examined different arbitration methods for the shared bus of a Java real-time processor and TDMA arbitration resulted as the most predictable method. This tendency can be further supported by the long-standing work on the *Time-Triggered Architecture (TTA)* [6], where timing-predictable communication is implemented by customized TDMA schedules. The TTA and other works also define Networks-on-Chip (NoCs) which use TDMA to provide guaranteed delays. With minor adaptions, the analysis techniques presented in this paper are also applicable to such NoCs.

The adaptation of multicore hardware to exhibit better timing properties is related to our work, since we also propose a configurable arbiter for the shared resource, but we assume standard components for the rest of the system. In other works, architectures which can explicitly produce a measurable WCET [10] or which are at least highly predictable [7] are proposed. Compared to our architecture these approaches require more changes to the hardware.

3 System Model

The system architecture that we assume is sketched in Figure 1. It is built according to the PROMPT guidelines [2], and thus is composed from n fully timing-composable ARM7TDMI cores. Each core has local scratchpads for instructions and data and is connected to the shared memory via a shared bus with configurable arbitration logic. The shared memory finally holds RAM memories for instructions and data and a boot ROM from which the cores read their packed binaries during system startup. We have deliberately not included caches in the architecture to be able to focus on the effects of the employed arbitration methods. Otherwise, imprecisions in the cache analysis might be able to affect the results for the shared bus. Nevertheless, caches can be easily integrated into the analysis framework, since we follow the standard approach from [16]. The whole system was modeled in the cycle-accurate virtual prototyping IDE CoMET [4] to be able to perform detailed measurements. A custom implementation of the bus arbitration logic has been designed, to be able to track the bus utilization and the imposed access delays in detail.

All implemented arbitration methods share the assumptions that bus transactions are uninterruptible and that a maximum duration m_{max} of an access to any device behind the

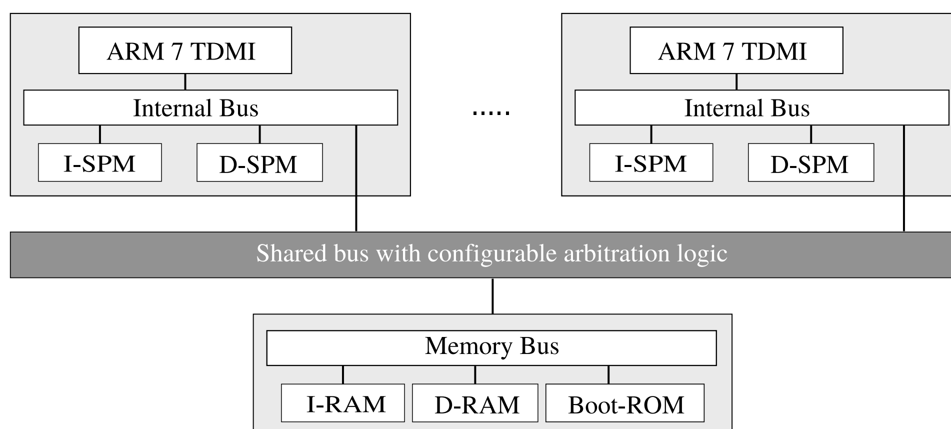


Figure 1 The employed system architecture (simplified).

bus is known. For simplicity of presentation we also assume that the bus runs at the same clock as the cores, but the offset analysis sketched in Section 4 can easily be extended to include a bus clock which is slower than the core clock, since it is tracking the schedule position in (higher precision) core ticks.

Fair arbitration (FAIR), also called *Round-Robin*, rotates the bus access among all cores. It maintains an *active core* $c_a \in \{0, \ldots, n-1\}$. When an access finishes c_a is advanced to the next core which requests the bus. Thus, each core can acquire the bus after at most $n-1$ other cores have performed their accesses.

Static priority-based arbitration (PRIO) assigns a unique priority $p_i \in \{1, \ldots, n\}$ to each core c_i and when there are multiple requests only the request from the core with the highest priority is granted. Nevertheless, since accesses are uninterruptible even the highest-priority core may have to wait until an ongoing transaction is completed.

TDMA creates a schedule consisting of n slots of size l and assigns an owner core $o_i \in \{0, \ldots, n-1\}$ to each slot. The current position in the schedule is determined by taking the current clock tick modulo nl. In each slot i only the owner is granted access to the bus and only in the interval $[il, \ldots, (i+1)l - m_{max}]$. The subtraction of m_{max} is necessary to make sure that accesses complete before the next slot begins.

Priority division (PD) is a generalization of TDMA. Instead of assigning an owner o_i it assigns unique priorities $p_{ij} \in \{0, 1, \ldots, n\}$ for each slot i and each core j. The bus is granted to the requesting core with the highest positive priority, only those with priority 0 are excluded from arbitration (this can be used to emulate TDMA behavior).

4 Analysis Framework

The general analysis framework is depicted in Figure 2. The analysis begins with a CFG reconstruction, which is based on pattern matching, and then does a simple register value analysis to identify memory access targets of load/store instructions. The main component, which is influenced by the choice of the arbitration policy, is the microarchitectural analysis. This stage computes bounds on the execution time of basic blocks. We sketch its architecture for the case with caches to show how caches can be integrated, even though our target architecture currently does not contain caches (see Section 3). The microarchitectural analysis associates an abstract pipeline state with each node in the CFG. Each abstract pipeline state is complemented by an abstract cache and bus state. The pipeline analysis is the driver of this stage: It simulates the possible processor actions on the abstract states and sends a request to the cache and bus stages whenever a memory access is performed (solid gray arrows in Figure 2). This request contains information about the target of the access and the timing of the access relative to the begin of the block's execution. If caches are present then the cache analysis may forward the request to the bus analysis, depending on whether the access is guaranteed to hit the cache or not. Similarly, the bus analysis receives the request together with its associated timing information and must decide how long it may take for this request to be granted the bus. We base this analysis on local abstract state information only, i.e. the bus analysis may not assume information about concurrently occurring accesses performed by other cores [1]. This means that for the worst-case we have to assume that concurrent accesses are occurring all the time. The bus and cache analyses must

[1] If concurrently occurring accesses are to be considered, a *Parallel Control Flow Graph* is needed together with analysis techniques which guarantee to cover all possible instruction and pipeline stage interleavings. These techniques, though principally possible, are prohibitively expensive since the enumeration of all possible interleavings leads to a state explosion [9].

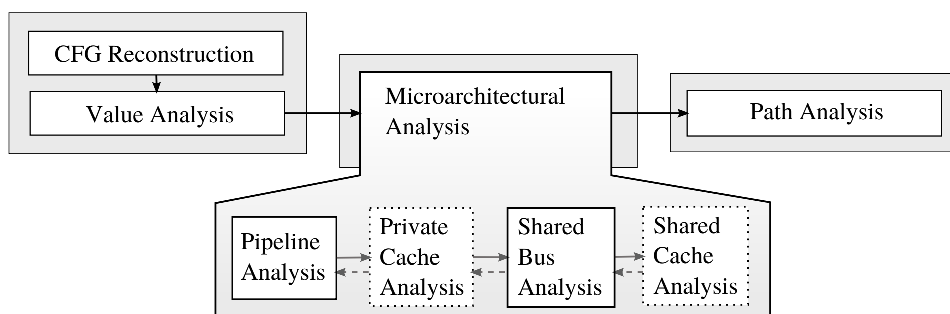


Figure 2 Analysis stages and their interaction.

then update the cache and bus state which is associated to the current abstract pipeline state and return an approximation of the timing behavior of the memory access to the pipeline analysis (dashed gray arrows in Figure 2). The updates of the basic block states (*transfer functions*) are integrated into a data-flow fixpoint iteration which converges in a safe approximation of the WCET for single executions of basic blocks (see [16]). This information is then used in the *path analysis* to compute the longest (shortest) path through the program whose length is the WCET (BCET). In the following we examine the abstract state and transfer functions for the shared bus analysis in more detail.

The *abstract bus state* for a basic block b as introduced in [5] is the set of *TDMA offsets* $O_b^{in} \subseteq \{0, \ldots, nl-1\}$, i.e. positions in the TDMA schedule with which the block execution may start. The bus analysis then computes multiple intermediate bus states O_b^i and finally a result state O_b^{out} that denotes the offsets after the block execution is finished. Initially, O_b^0 is set to O_b^{in}. The pipeline analysis then repeatedly hands accesses $a_i \in b, i \in \mathbb{N}$ to the bus analysis, together with an execution time set $T_{a_i} \in 2^{\mathbb{N}}$ that bounds the time that passed since access a_{i-1} (or the block start for $i = 1$). The bus analysis must then determine the set of possible memory access times D_{a_i} that the shared memory may need to serve this access. With these values the bus analysis computes $O_b^i = \mu_c(O_b^{i-1}, T_{a_i}, D_{a_i})$, where c is the core which issued request a_i. Also it must return a set of possible execution times $\delta_c(O_b^{i-1}, T_{a_i}, D_{a_i})$ to the pipeline analysis to describe the resulting timing for a_i.

By considering each input offset in separation we can define $\mu_c(O, T, D) = \bigcup_{o \in O, t \in T} \{\Phi_c(o + t \mod nl, D)\}$. $\Phi(o, D)$ determines the resulting offsets for an access at the offset o whose runtime is bounded by D.

$\Phi : \mathbb{N} \times 2^{\mathbb{N}} \rightarrow 2^{\mathbb{N}}$ must be defined for each arbitration method, to reflect the timing of the specific method. For TDMA and PD we define s_o as the slot containing offset o and s_c as the first slot after offset o where core c has maximum priority or which is owned by c. $s_c = s_o$ is explicitly allowed. With these definitions we have

$$\Phi_c^{TDMA}(o, D) = \begin{cases} \{o\} \oplus D & \text{if } o \in \omega(s_c) \\ \{s_c l\} \oplus D & \text{if } else \end{cases} \tag{1}$$

The first case in Equation 1 models an access inside slot s_c and the second case handles accesses outside of it. The operator $\oplus$ is defined by $\oplus(X, Y) = \{x + y | x \in X, y \in Y\}$ and $\omega(s_c) = \{s_c l, \ldots, (s_c + 1)l - m_{max}\}$ is a shorthand for the "grant window" of offsets inside

s_c where an access from core c will be immediately granted.

$$\Phi_c^{PD}(o, D) = \begin{cases} \{o\} \oplus \{0, \dots, m_{max}\} \oplus D & \text{if } o \in \omega(s_c) \\ \left(\bigcup_{s_i \in \{s_o, \dots, s_c - 1\}} \phi_c(s_i, D) \right) \cup \Phi_c^{TDMA}(o, D) & \text{if } o \notin \omega(s_c) \\ \emptyset & \text{if } \nexists s_c \end{cases} \tag{2}$$

$$\phi_c(s, D) = \begin{cases} \{sl, \dots, (s+1)l - m_{max}\} & \text{if } p_{sc} > 0 \\ \emptyset & \text{else} \end{cases} \tag{3}$$

The case structure for Φ_c^{PD} in Equation 2 is similar to the one for Φ_c^{TDMA}, but with the additional case that c is not the top-priority core in any slot (case three). Also it has to account for possibly ongoing transactions, even in slots in which c has the highest priority (case one). For accesses outside of s_c (case two) $\phi_c(s, D)$ as given in Equation 3 contributes all offsets which may result from the access being granted in a successive slot where c has positive but not the highest priority.

For fair and priority-based arbitration we can only supply conservative bounds, since these methods require knowledge about all possibly concurrently occurring transactions, which is hard to obtain in general, as noted above.

$$\Phi_c^{FAIR}(o, D) = \{o\} \oplus \{0, \dots, nm_{max}\} \oplus D \tag{4}$$

$$\Phi_c^{PRIO}(o, D) = \begin{cases} \{o\} \oplus \{0, \dots, m_{max}\} \oplus D & \text{if } \forall i \in \{1, \dots, n\} : p_i \leq p_c \\ \emptyset & \text{else} \end{cases} \tag{5}$$

Therefore, Equations 4 and 5 are only stating that FAIR accesses may experience 0 to nm_{max} cycles delay. For PRIO accesses the two listed cases are symmetric to case 1 and case 3 in Equation 2 since PRIO is just a specialization of PD.

It is easy to extend the definition of Φ such that it does not only return the resulting offset but also the time that it took to perform the arbitration and the access. With this extension we can define δ_c similar to μ_c. In cases in which the original Φ returned an empty set, the runtime is defined to be $\{\infty\}$, which happens for accesses which have no bounded duration (e.g. PRIO).

As mentioned before, the μ_c and δ_c functions are then used in the transfer function of the microarchitectural analysis stage. The join function, which is used when control flow from at least two different predecessors joins at a basic block, is simply the set union of the incoming offset sets.

5 Evaluation

Due to the lack of standard multicore real-time benchmarks, we chose to execute independent tasks from the MRTC, UTDSP, MiBench, MediaBench and DSPstone benchmark suites on the single cores, amounting to 110 flow-fact-annotated, independent *benchmark tasks* in total. In the experiments, we grouped together benchmarks with similar runtime and executed *packages* with one benchmark per core. The packages were formed by sorting the benchmarks in the order of their single-core ACET and then having a window of size 1/2/4/8 slide over this list, collecting all 110/109/107/103 possible combinations. All cores start their assigned task synchronously and execution finishes when all tasks have been completed. Thus, since the benchmarks have different runtimes there will be some amount of inevitable *completion time jitter*, but apart from that this scenario models a system with high load. All

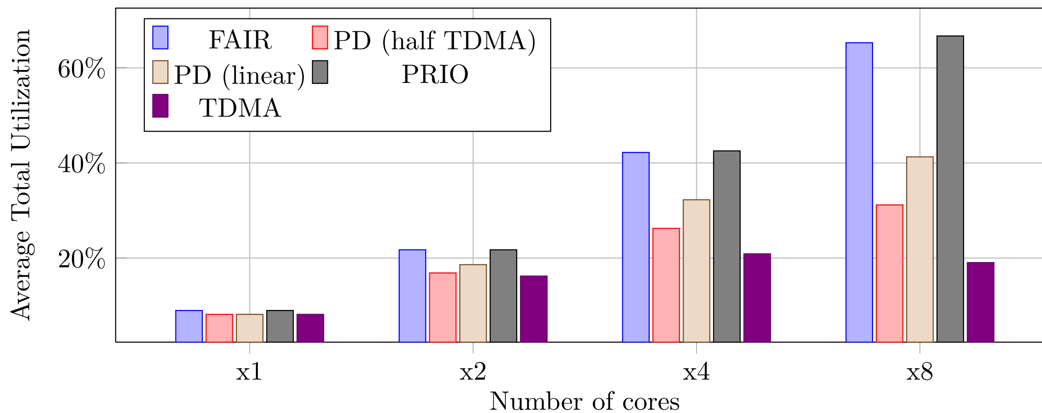


Figure 3 Average total bus utilization for different platforms.

the benchmarks read their inputs and store their outputs in the shared memory, whereas all program code as well as the stack was allocated in the scratchpad memory of the individual cores. This emulates the (reasonable) scenario that I/O is done via a shared device, whereas code and local data are kept in local memories for performance reasons. Also, all benchmarks were compiled with moderate optimization (optimization level O1). The memory access durations were set to 1 cycle for the scratchpads and $m_{max} = 3$ cycles for the shared memory. For most data-processing instructions the ARM7TDMI needs only 1 cycle, branches need 3 cycles and multiply instructions may need up to 5 cycles.

Concerning the parametrization of the arbitration methods we have selected simple heuristics to demonstrate some key impacts. For *PRIO* the priorities were assigned such that $t_i > t_j \Leftrightarrow p_i > p_j$ where t_i is the single-core runtime of the task mapped to core i. We use this strategy, also known as "largest job first", here to speed up long-running tasks and thus to decrease the completion time jitter. For *TDMA* (and also for *PD*) we set the slot size $l = m_{max}$ to keep delay times as small as possible. Our experiments have shown that higher slot lengths impose both higher WCET and ACET values. Also, for *TDMA* we set $o_i = i$ such that each core owns a single slot. For *PD (linear)* each slot i is "owned" by core i by setting $p_{ii} = n$. Priorities for all other cores are distributed in the same way as for *PRIO* i.e. in the order of single-core task runtime. The variation *PD (half TDMA)* for all slots $i \in \{0, \dots, \lfloor n/2 \rfloor\}$ assigns $p_{ii} = 1$ and $\forall j \neq i : p_{ij} = 0$, thus effectively making these slots pure TDMA slots. All slots $i \in \{\lfloor n/2 \rfloor + 1, \dots, n\}$ are configured in the same way as for *PD (linear)*.

Figure 3 shows the average[2] utilization resulting for different values of n. As expected, *FAIR* and *PRIO* show superior utilization that scales linearly with the number of cores, since these are *work-conserving* arbitration methods, i.e. as long as there are active requests they do not insert wait cycles. *TDMA* shows some increase in utilization with rising n, but it is stagnating at around 20% due to slots which remain unused by their owners. For $n = 8$ the utilization is actually *decreasing* again below 20%. *PD* is also not work-conserving, since it must delay requests when they cannot be served in the current slot, which may happen frequently for our setting of $l = m_{max}$. Still both *PD* configurations show a linear increase in utilization, with *PD (half TDMA)* being slightly behind *PD (linear)*. For *PD (linear)* the utilization is twice as high as for *TDMA*, which is also reflected in the average ACETs of the

[2] Since we only report relative values here, we use *average* as a synonym for the *geometric mean*.

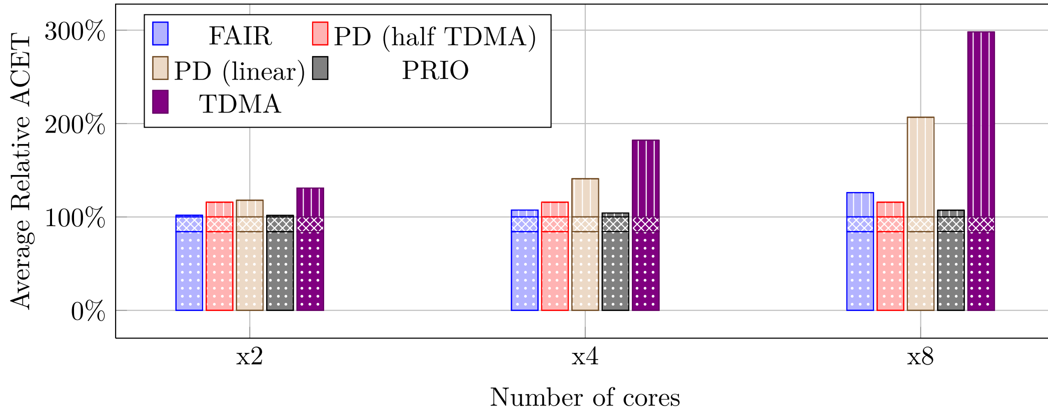


Figure 4 Average relative measured execution time (ACET) for different platforms (Baseline = execution time on single-core platform).

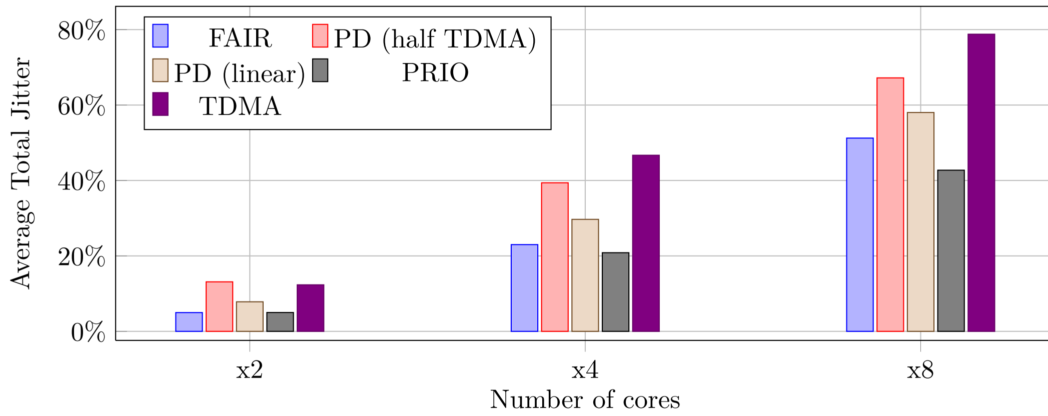


Figure 5 Average benchmark execution time jitter for different platforms.

benchmarks as shown in Figure 4.

In general, the ACET per task is inversely proportional to the achieved utilization values. The dotted areas in Figure 4 show the portion of the ACET which is used for computation and local memory accesses (stack and program code, see Section 3), the crosshatched areas show the portion in which the task is *using* the shared bus and the areas with vertical bars show the percentage of the ACET in which the task is *waiting* for the shared bus. For *TDMA* it becomes visible that e.g. for the configuration with 8 cores, the tasks are on average using more cycles for waiting than for performing computation and actual memory accesses.

The ACET and utilization values are influenced by the completion time jitter of the benchmark packages, that is the length of the time interval between the first termination of a benchmark on any core and the termination of the last benchmark. Especially for *TDMA* the jitter is problematic since it leaves slots of already terminated cores unused. Figure 5 shows the jitter as a percentage of the total runtime of the benchmark package (i.e. the runtime of the longest benchmark). It is visible that the low utilization values for *TDMA* are to some extent related to the rising jitter, but since this increase is itself triggered by the usage of *TDMA* this is an inherent drawback of the policy.

Finally, Figure 6 shows the average of the quotient of single-task WCET and single-task ACET, which is a bound on the maximum possible WCET overestimation. The highest

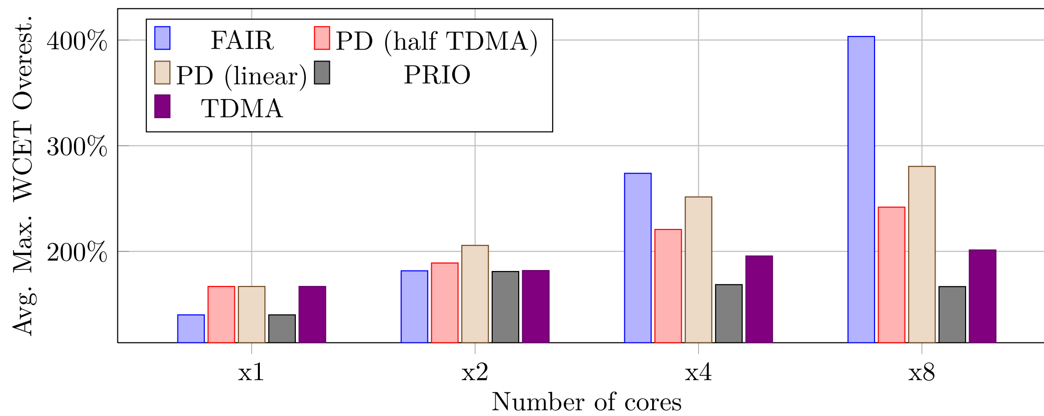


Figure 6 Average maximum task WCET overestimation for different platforms. Since the baseline corresponds to the measured ACET on the respective platform, the values represent maximum overestimation ratios.

WCET increases are attained by *FAIR* since it must always assume a worst-case delay of $(n-1)m_{max}$ cycles, but up to 2 cores *FAIR* is still competitive. Concerning the *PRIO* results in Figure 6, they are almost constant between the different configurations since here only the tasks for which a WCET can be determined at all are considered. This in all cases is only the highest-priority task whose WCET is increased by 40% to 81% on average, due to the presence of uninterruptible lower-priority accesses. *TDMA* shows the smallest overestimation ratio, but it is notable that both *PD* approaches are following very closely after *TDMA* in the WCET ranking. The small increase in WCET for e.g. *PD (linear)* is compensated by far better ACET and utilization values which makes *PD* a very appealing method for mixed-criticality systems.

6 Summary & Future Work

Current multi-core processors are mainly not timing-predictable due to a number of reasons, with one of them being accesses to shared resources. Since some amount of sharing is inevitable in multi-core systems we have demonstrated advantages and disadvantages of arbitration schemes for shared resources. Therefore, this work can serve as a basis for selecting a suitable arbitration policy, depending on the needs of the platform. For the first time, the drawbacks of TDMA in terms of average-case performance were quantified in comparison to other arbitration methods and it was shown, that priority division is a promising alternative for systems running mixed-criticality workloads, since it allows the fine-grained trading of ACET and bus utilization vs. WCET.

In the future, we would like to examine the effects of software optimizations to the achievable prediction accuracy, e.g. by grouping bus accesses or by restructuring tasks into read, execute and write phases. Another interesting perspective is the optimization of bus schedule parameters and the refinement of the presented multi-core WCET analysis.

7 Acknowledgments

We would like to thank Synopsys for the provision of the virtual prototyping IDE CoMET.

References

1. Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proc. of SCOPES*, 2010.
2. Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.
3. Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures using UPPAAL. In *Proc. of WCET*, July 2010.
4. Synopsys Inc. CoMET system engineering IDE. `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/CoMET-METeor.aspx`.
5. Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *ECRTS*, pages 3–12, 2011.
6. Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
7. Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance. In *Proceedings of the International Conference on Computer Design*, ICCD 2012, October 2012.
8. Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the Real-Time Systems Symposium*, RTSS '10, pages 339–349, Washington, DC, USA, 2010. IEEE Computer Society.
9. Robert Mittermayr and Johann Blieberger. Timing Analysis of Concurrent Programs. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23, pages 59–68, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
10. Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Computer Architecture News*, 37(3):57–68, 2009.
11. R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 741–746, 2010.
12. Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Transactions on Embedded Computing Systems*, 2009.
13. Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proc. of WCET*, 2006.
14. H. Shah, A. Raabe, and A. Knoll. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Proc. of DATE*, pages 1–4, 2011.
15. Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.
16. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.

Automatic WCET Analysis of Real-Time Parallel Applications*

Haluk Ozaktas, Christine Rochange, and Pascal Sainrat

IRIT – Université de Toulouse
France
firstname.lastname@irit.fr

Abstract

Tomorrow's real-time embedded systems will be built upon multicore architectures. This raises two challenges. First, shared resources should be arbitrated in such a way that the WCET of independent threads running concurrently can be computed: in this paper, we assume that time-predictable multicore architectures are available. The second challenge is to develop software that achieves a high level of performance without impairing timing predictability. We investigate parallel software based on the POSIX threads standard and we show how the WCET of a parallel program can be analysed. We report experimental results obtained for typical parallel programs with an extended version of the OTAWA toolset.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, parallel programming, thread synchronisation

Digital Object Identifier 10.4230/OASIcs.WCET.2013.11

1 Introduction

Future real-time embedded systems will have to follow the global trend towards multicore computing units, which is mainly guided by power efficiency considerations. Designing time-predictable multicore architectures is at the heart of several research projects, e.g. T-CREST[1] and parMERASA[2]. Now, when hardware solutions are available, software will have to be carefully designed to optimise the usage of resources. In some cases, the target is a high task throughput: it can be achieved by co-scheduling independent tasks on the cores. Other applications, e.g. command-control functions in cyber-physical systems, instead require shortened response times. For some of them, that exhibit intrinsic data or control parallelism, the execution time of individual tasks can be reduced by applying parallel programming techniques: a task is decomposed into threads that are run in parallel, each of them processing one part of the workload. In this paper, we focus on this class of programs.

Several parallel programming paradigms can be considered depending on the problem decomposition (task- or data-parallelism) and on the way threads can communicate, which is highly related to the target hardware architecture. Various programming languages and APIs can be used to develop parallel programs. We focus on the POSIX threads standard which is widely used in the industry.

In real-time systems, special attention must be paid to task scheduling: it must be guaranteed that critical tasks will meet their hard deadlines in any situation. Real-time

* The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

[1] www.t-crest.org
[2] www.parmerasa.eu

© Haluk Ozaktas, Christine Rochange, and Pascal Sainrat;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 11–20

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

task scheduling is a hot research topic. All the proposed strategies rely on estimations of the worst-case execution times (WCET) of critical tasks. Several approaches have been proposed in the past to determine WCET upper-bounds considering sequential tasks running on uni-processors. If these techniques are still to be used when independent tasks run on time-predictable multicores, specific solutions have to be developed for parallel applications, composed of synchronising threads. This is the purpose of this paper.

The paper is organised as follows. In Section 2, we discuss the scope of our work and we give an overview of related work. Section 3 introduces our approach to the automatic WCET analysis of POSIX-based parallel tasks. Experimental results are reported in Section 4 and Section 5 concludes the paper.

2 Scope of the paper and related work

2.1 Time-predictable multicores

Various classes of parallel architectures exist, from chip multicores to clusters and grids of computers, or from general-purpose processing units to specialised accelerators like GPUs. However, to be considered as candidates to build hard real-time systems, these architectures should enforce timing analysability: it should be possible to compute the WCET of a critical task running in parallel with other tasks. The main difficulty comes from inter-task conflicts: they make the latencies of accesses to shared resources hard to predict. This mainly concerns shared caches, memory controllers and interconnection networks.

Two kinds of approaches have been proposed. A first group of solutions consist in considering all together the tasks that might be running at the same time in order to estimate their possible interactions and their impact on the worst-case execution times. This strategy has been considered for the analysis of shared caches [9, 11, 14] and shared busses [2, 20]. A second class of approaches aim at designing hardware that enforces spatial and timing isolation for critical tasks. Cache locking and partitioning schemes [21, 16] belong to this category. Timing isolation can also be supported by appropriate arbitration mechanisms, e.g. for a shared bus [19, 10] or a memory controller [1, 15]. More globally, several European projects have been launched to design time-predictable multicores, e.g. MERASA [22], PREDATOR [4], T-CREST and parMERASA.

In the following, we assume we have a time-predictable shared-memory multicore architecture and the related WCET analysis tool capable of analysing sequential applications: modelling hardware-level thread interactions is out of the scope of our work.

2.2 Real-time and WCET-aware parallel applications

Various parallel programming models exist and are supported by a large number of programming languages and APIs. In this work, we focus on applications developed on the widely-used POSIX threads standard.

Programs written with POSIX threads are characterised by explicit thread control (creation and join) and explicit thread synchronisation through mutexes, condition variables and barriers. Figure 1 shows a sample program that we will use as a running example in the paper. Determining the WCET of such a parallel program comes up to computing the WCET of the main thread, taking into account the costs for thread control and thread synchronisations. The predictability of these costs highly depend on the implementation of the system software. This is discussed in Section 2.3.

```
int main() {
  for (int i=0; i<2; i++)
    CREATE_THREAD(&work);
  ...
  BARRIER(&bar,3);   // ID=bar
  ...
  for (int i=0; i<2; i++)
    JOIN(i+1);   // ID=join
}
```

```
void work() {
  ...
  BARRIER(&bar,3);   // ID=bar
  ...
  MUTEX_LOCK(&lock);   // ID=cs
  ... // critical section
  MUTEX_UNLOCK(&lock);   // ID=cs
  ...
}
```

Figure 1 Example code.

Real-time parallel applications should be designed with time predictability in mind. As we will see later, stall times at synchronisations impact the WCET. Then the main recommendations come from the way these stall times can be estimated: synchronisation operations as well as the involved threads should be easy to identify. As a consequence, the number of threads should be statically fixed and synchronisation patterns should make it possible to determine how long one thread may be stalled by another one. The latter can be achieved by using standard synchronisation patterns, like critical sections and barriers.

In the following we also consider that the number of threads is lower than or equal to the number of cores so that all the threads can execute in parallel, each on a different core. In practice, it could be accepted that the number of threads exceed the number of cores. In such a case, however, the scheduling of threads and their mapping to cores must be decided statically [17]. This way, the timing analysis can determine how to compose their individual WCETs. This option is not considered in the paper.

2.3 Time-predictable system software

The analysis of stall times requires the synchronisation to be implemented with time-predictable primitives. Mainly, these primitives should allow upper bounding the stall time of a thread at a synchronisation. Ticket locks should, for example, ensure that threads reaching a critical section will be granted access in a First-Come First-Served fashion. The design of such predictable primitives is discussed in [23, 5]. We assume that the applications are developed using such routines. In addition, timing analysis either needs an upper-bound in the latency of a thread creation or a hardware mechanism that enforces a synchronous start of created child threads.

2.4 Related work

As mentioned earlier, using multicores to build hard real-time systems is not common yet. Research on WCET analysis on multicores has essentially focused on the predictability of accesses to shared resources, as overviewed in Section 2.1. There have been very few contributions to the analysis of parallel programs. The timing analysis of a parallelised control-loop style application was reported in [6]. In [18], a first attempt to *manually* compute the WCET of an industrial parallel program with static analysis techniques was reported. Individual code segments were analysed using the OTAWA toolset, then their WCETs were combined outside the toolset by hand to determine the WCET of the whole application.

In [7], the authors propose a method based on timed automata to model the behaviour of a parallel program. Model checking techniques are used to determine the WCET of the whole program by verification. In [8], they consider a simplified parallel programming language

and introduce an approach based on abstract interpretation to perform simultaneous timing analysis of the different threads. The predictability of various parallel programming models, e.g. GPU and data parallel programming, is investigated in [13].

3 Approach to the WCET analysis of parallel applications

The execution time of a parallel program is the execution time of its longest thread. In our model, the main thread creates child threads and later joins them. Then determining the WCET of a parallel program comes to computing the WCET of its main thread. This time is impacted by the child threads:

- The latency of the thread creation operation must be accounted for;
- The main thread may have to wait for other threads when it reaches a barrier or the lock acquisition operation before a critical section. The worst-case stall time must be estimated.
- When joining the child threads, the main thread has to wait for their termination

3.1 Timing analysis of synchronisations

We distinguish two kinds of synchronisations: critical sections, guarded by locks, and progress synchronisations, implemented by barriers or conditions (wait and signal). In both cases, a thread that reaches a synchronisation primitive may be forced to wait before proceeding. Its worst-case stall time (WCST) must be estimated.

3.1.1 Worst-case stall times

Critical section

Entering a critical section is typically achieved by acquiring a lock. If no other thread requests the lock at the same time, then the synchronisation does not generate any stall. But in the worst case, all possible contenders try to acquire the lock simultaneously, and the thread has to wait for all other threads to release the lock (provided locks are granted in a First-Come First-Served fashion). This is illustrated in the left-side part of Figure 2.

The WCST at the critical section for the leftmost thread, denoted by S, is computed assuming the two other threads already have requested the lock. Then S is the sum of the times during which each of them holds the lock, i.e. the WCETs of their critical sections:

$$S = w_1 + w_2$$

Progress synchronisation (barrier)

The right-side part of Figure 2 illustrates the stall time of a thread at a barrier. The WCST is determined by considering the previous collective synchronisation point, i.e. the previous point where all the involved threads did synchronise before the barrier.

The (actual) stall time of one thread (thread i) at a barrier to be reached by a single other thread (thread j) would be given by $\max(0, t_j - t_i)$, where t_i and t_j are the actual execution times of threads i and j to reach the barrier from the previous synchronisation point: either $t_i \geq t_j$ and thread i does not have to wait, or the stall time is the difference between their execution times.

Now, threads generally exhibit variable execution times: $t_i \in [b_i, w_i]$ where b_i and w_i are the best- and worst-case execution times for thread i (similarly, $t_j \in [b_j, w_j]$). Then attention should be paid to how the difference between their execution times is computed.

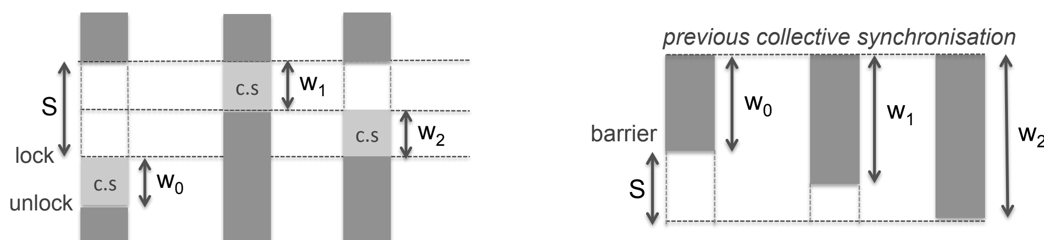


Figure 2 Stalls due to synchronisations.

Theoretically, the longest stall time by thread i when $t_i < t_j$ is given by $w_j - b_i$ (difference between the worst-case execution time of thread j and the best-case execution time of thread i). However, computing the WCST for thread i is done in the context of determining its WCET. As a result, the worst-case value for $t_j - t_i$ is computed as $w_j - w_i$.

Generalising to several threads, as in the example shown in Figure 2 (right side), we get:

$$S = \max(0, (w_1 - w_0), (w_2 - w_0))$$

3.1.2 Abstract view of synchronisation primitives

While a synchronisation operation is simply seen as a call to a system-software primitive, things are a bit more complex from the point of view of WCET analysis which is done at cycle-/instruction-level. The main issue is to identify key locations in the code of the primitives: the point where a thread may be stalled and the point where a thread may signal other threads (allowing them to resume their execution). Finding out these locations is a hard task and having it done automatically is still challenging. This is the reason why we need the parallel application to use known primitives, that have been previously analysed manually (as described in [18]). We plan to release this constraint by designing a specific format to describe synchronisation routines, so that the user could use his own primitives and provide a description for them.

3.1.3 Computation of the global WCET

The WCET of the whole application is computed as the WCET of the main thread to which WCSTs at synchronisations are added. In our example (see Figure 3), two WCSTs must be estimated for the main thread. The first one, S_1, is related to a barrier. As explained in Section 3.1.1, it is determined considering the threads' WCETs from the previous collective synchronisation, which is the creation of the child threads, to the barrier (we assume that the cost of thread creation is known). The second stall time, S_2, at the join with child threads, can be analysed similarly. The previous collective synchronisation is the barrier. However, the code executed by the child threads from the barrier to the exit includes a critical section. Then S_2 depends on S_3, which can be determined as shown earlier.

Figure 5 depicts the global procedure to perform the timing analysis of a parallel program. First, synchronisation patterns must be identified. This may be a complex task. To make it simpler, we rely on user-provided annotations that we will describe in Section 3.2. The second step determines the dependencies among the stall times and builds a WCST tree, as the one shown in Figure 4. This is done from the root down to the leaves: a branch ends when a WCST can be computed from partial execution paths that do not include any synchronisation. WCSTs can then be estimated by climbing up the tree from the leaves to the

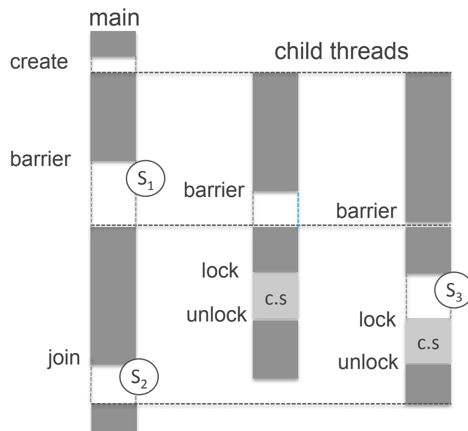


Figure 3 Example program.

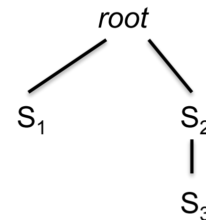


Figure 4 WCST tree.

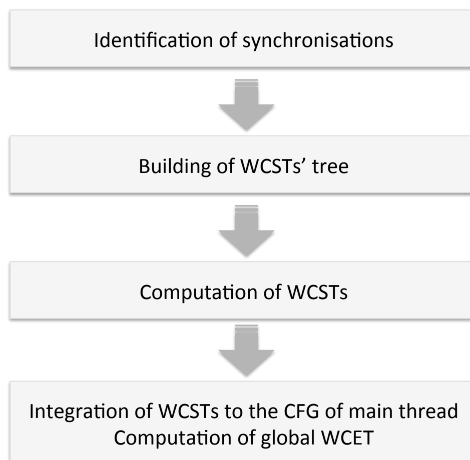


Figure 5 Global procedure.

```
<barrier id="bar">
  <thread id="0-2">
    <last_sync ref="BEGIN"/>
  </thread>
</barrier>
<csection id="cs">
  <thread id="1-2"/>
</csection>
<sync id="join">
  <thread id="0">
    <wait id="1-2">
      <sync ref="END"/>
      <last_sync ref="bar"/>
    </wait>
  </thread>
</sync>
```

Figure 6 Annotations for the example code.

root. They are added to the WCETs of the corresponding basic blocks in the program CFG. The final stage integrates the WCSTs into the ILP formulation of the WCET computation (IPET method [12]).

3.2 Annotations of parallel programs

To help the analysis of parallel programs, and in particular of their synchronisations, we have designed an annotation format. It can be used to provide information on synchronisation patterns. The annotation format includes two parts:

- A set of identifiers annotated in the source code, to allow further reference to specific points in the program, i.e. calls to synchronisation primitives. Identifiers are specified as C comments (`// ID=...`), as can be seen in the example code (see Figure 1).
- Additional information, e.g. the threads involved in a synchronisation, are provided in a separate XML-based file. Some elements of this file are described below[3].

[3] Due to space limitations, only a subset of our annotation language is described in this paper. The full language supports more complex synchronisation patterns.

Figure 6 shows the annotation file that describes our example code. It specifies three synchronisations that are likely to generate stall times: a barrier, a lock-based protection for a critical section, and joining the child threads for the main thread.

The `barrier` element refers to a barrier identifier put in the source code. The inner `thread` element indicates which threads should meet at the barrier (`0` is the main thread). For these threads, the previous collective synchronisation is specified in the `last_sync` element, with the `BEGIN` built-in value that refers to the start of each thread. The `csection` element provides details on the synchronisation at the entry of the critical section. The inner `thread` element shows that the two child threads may compete for the lock. Finally, the `sync` element refers to the join operation executed by the main thread as specified by the nested `thread` element. The innermost `wait` element indicates that it should wait for threads 1 and 2 to reach the end of their execution (as specified with the built-in value `END`).

4 Experiments

4.1 Methodology

Our solution to automatically analyse the WCET of parallel programs helped with user-provided annotations has been implemented on top of the OTAWA toolset [3]. OTAWA provides an API to build WCET computation tools based on static analysis techniques. We have extended the library with utilities to parse annotation files, to retrieve synchronisations in the binary code, to build the WCST tree, to analyse the WCSTs and then to integrate them in the linear program used to determine the global WCET.

Since we focus on software interactions, we have considered a simple architecture in which each instruction executes in a single cycle with a configurable additional latency for memory accesses. We have found that the results presented below do not depend on the value of the latency (raw values do, but not the shape of curves).

4.2 Benchmarks

We have analysed two different parallel implementations of a kernel solving a partial differential equation on a 2D-grid. The first version uses the iterative Gauss-Seidel method where each point is computed based on its immediate north and west neighbours. There is no dependency between the points belonging to the same anti-diagonal: they can be computed in parallel. However, dependencies among anti-diagonals should be respected. The algorithm iterates until convergence. Our parallel implementation first divides the grid into compartments such that the main anti-diagonal has the same number of compartments as the number of threads. It exploits the independence of the compartments within a same anti-diagonal. This implementation includes three barriers and one critical section. The main thread participates in the computation and execute the same function as the child threads.

The second version implements the Jacobi method where each point can be computed independently of other points: this improves the intrinsic parallelism but generally requires a larger number of iterations to converge. Our parallel implementation of this algorithm assigns a block of lines to each thread. Threads execute in parallel within an iteration. The code contains two barriers and one critical section. For both methods, we have defined a maximum number of iterations in order to be able to compute a WCET value.

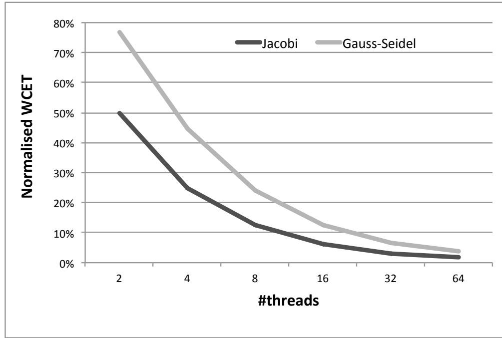


Figure 7 Normalised WCET.

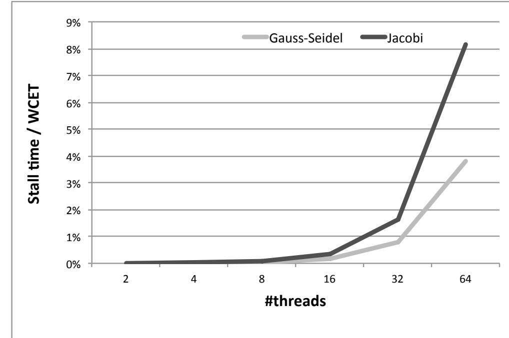


Figure 8 Impact of stall times.

	Gauss-Seidel	Jacobi
2	0.559	0.379
4	1.046	0.705
8	2.177	1.446
16	3.718	2.679
32	8.796	5.782
64	17.999	11.855

Figure 9 Computation times in seconds.

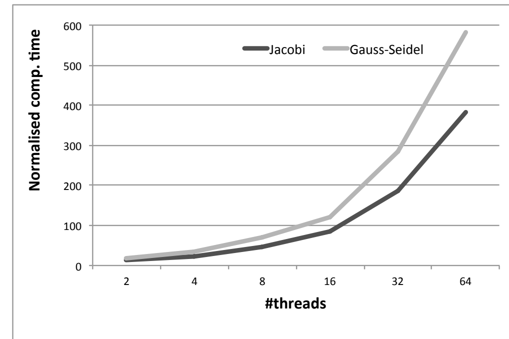


Figure 10 Normalised computation times.

4.3 Results

We report experiments carried out for the two algorithms described above, considering both sequential and parallel (from 2 up to 64 threads) versions.

Figure 7 shows the WCETs of parallel implementations normalised to the WCET of the sequential code. For the same number of threads, the Jacobi algorithm gets higher speed-ups than the Gauss-Seidel method: this was expected since there is no dependence between points in Jacobi method which yields to higher parallelism.

Figure 8 plots the contribution of stall times to the WCET of the application. For up to 32 threads, the impact of worst-case stall times is negligible. For 64 threads, stall times contribute from 4% (Gauss-Seidel) to 8% (Jacobi) of the WCET. These low contributions are mainly due to the fact that all the threads run the same code. Then their worst-case arrival times at barriers are equal. Thus the stall times are only due to the critical section. They rapidly increase with the number of threads because, in the worst case, a thread is stalled until all the possible contenders execute the critical section and release the lock. This shows the importance of limiting the number of contending threads to optimise the WCET.

Figure 9 provides the raw values of the computation time (in seconds) of the automatic WCET analysis of the parallel codes. In Figure 10 these times are normalised to the WCET computation time of the sequential version (which is 0.031 seconds). Analysing a parallel application is noticeably longer than analysing its sequential version. This was somewhat expected since the WCET estimation of a parallel program requires many small WCET analyses on partial paths. Now, in these experiments, the WCET was analysed as if the threads did execute different functions, to reflect a pessimistic situation. In our

two benchmark codes, all the threads instead share the same function. As a result, the real computation cost would be that of the parallel program with two threads (the main and one child), i.e. about 18 times the computation cost for the sequential version for the Gauss-Seidel algorithm (about 12 times for the Jacobi method).

5 Conclusion

With the emergence of multicore architectures in the embedded systems market, one strategy to get high computing power will be to parallelise software. Now, for hard real-time systems, timing predictability is a key issue. It requires specific solutions at the hardware level, since interactions among concurrent threads must be controlled in some way to make their timing analysis possible. This point is at the core of several terminated and ongoing research projects and was considered as solved in this paper. Parallel programming introduces software-level interactions between threads through synchronisation operations. These synchronisations engender stall times that must be accounted for when analysing the worst-case execution times of tasks. This is the problem we have tackled in this work.

We have introduced an approach for an automatic timing analysis of parallel applications. It consists in estimating the synchronisation-related stall times of each individual thread and in considering them as extra-costs for the associated basic blocks in the CFG. This way, the stall times are accounted for within the WCET computation process.

Determining the worst-case stall times due to synchronisations requires a detailed analysis of the synchronisation patterns and of the binary code of synchronisation primitives. To perform this task we rely on annotations that must be generated by the user. Once synchronisation operations are identified, WCSTs are recursively computed.

We have implemented our algorithm on top of the OTAWA library and experimented it on parallelised versions of the Gauss-Seidel and Jacobi algorithms. Experimental results show that the worst-case impact of synchronisation stalls on WCET estimates remains limited (8% for 64 threads). The cost of analysing a parallel code remains reasonable when all the threads execute the same function (around 12 to 18 times the computation cost of the sequential version) but rapidly increases with the number of threads when they run different codes.

As future work, we plan to apply our approach to larger applications and to analyse the impact of parallel programming patterns to the worst-case performance of programs. We will also investigate automatic extraction of synchronisation patterns from the binary code.

References

1. B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *5th Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2007.
2. B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. In *WiP of Real-Time Systems Symposium (RTSS)*, 2009.
3. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Workshop on Software technologies for Embedded and Ubiquitous Systems (SEUS)*, 2011.
4. C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Int'l Conf. on Embedded Real Time Software and Systems*, 2010.
5. M. Gerdes, F. Kluge, T. Ungerer, and C. Rochange. The split-phase synchronisation technique: Reducing the pessimism in the WCET analysis of parallelised hard real-time

programs. In *Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

6 M. Gerdes, J. Wolf, I. Guliashvili, T. Ungerer, M. Houston, G. Bernat, S. Schnitzler, and H. Regler. Large drilling machine control code—parallelisation and WCET speedup. In *Int'l Symp. on Industrial Embedded Systems (SIES)*, 2011.

7 A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using uppaal. In *Workshop on WCET Analysis*, 2010.

8 A. Gustavsson, J. Gustafsson, and B. Lisper. Toward static timing analysis of parallel software. In *Workshop on WCET Analysis*, 2012.

9 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium (RTSS)*, 2009.

10 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, 2011.

11 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symposium (RTSS)*, 2009.

12 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, 1995.

13 B. Lisper. Towards parallel programming models for predictability. In *Workshop on WCET Analysis*, 2012.

14 M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*, 2010.

15 M. Paolieri, E. Quiñones, F. J Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4), 2009.

16 P. Paolieri, E. Quiñones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Int'l Symp. on Computer Architecture (ISCA)*, 2009.

17 M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Design, Automation and Test in Europe (DATE)*, 2009.

18 C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F.šek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In *Workshop on WCET Analysis*, 2010.

19 J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium (RTSS)*, 2007.

20 S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation and Test in Europe (DATE)*, 2010.

21 V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Design Automation Conference (DAC)*, 2008.

22 T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5), 2010.

23 J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor. In *Int'l Conf. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2010.

Integrated Worst-Case Execution Time Estimation of Multicore Applications*

Dumitru Potop-Butucaru[1] and Isabelle Puaut[2]

1 **INRIA, Paris-Rocquencourt,** `dumitru.potop@inria.fr`
2 **University of Rennes 1/IRISA, Rennes,** `isabelle.puaut@irisa.fr`

Abstract

Worst-case execution time (WCET) analysis has reached a high level of precision in the analysis of sequential programs executing on single-cores. In this paper we extend a state-of-the-art WCET analysis technique to compute *tight* WCETs estimates of parallel applications running on multi-cores. The proposed technique is termed *integrated* because it considers jointly the sequential code regions running on the cores and the communications between them. This allows to capture the hardware effects across code regions assigned to the same core, which significantly improves analysis precision. We demonstrate that our analysis produces tighter execution time bounds than classical techniques which first determine the WCET of sequential code regions and then compute the global response time by integrating communication costs. Comparison is done on two embedded control applications, where the gain is of 21% on average.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases WCET estimation, multicore architectures, parallel programming

Digital Object Identifier 10.4230/OASIcs.WCET.2013.21

1 Introduction

Multi-core systems are becoming prevalent in both general purpose and embedded systems. Their adoption is driven by scalable performance arguments, but this scalability comes at the price of increased software complexity. Indeed, multi-core systems run parallel software involving potentially complex synchronizations between the sequential programs executed on the various cores. In the current state of the art of validation of real-time multi-task software, temporal validation is achieved by computing the worst-case response time (WCRT) of every task, defined as an upper bound for the duration between the task arrival and its termination. Two main classes of techniques, usually applied sequentially, are used: (i) Worst-case execution time (WCET) estimation, which works on sequential programs, and (ii) WCRT estimation, that computes response times thanks to WCET values as inputs.

WCET analysis emphasizes the importance of hardware micro-architecture. Indeed, in its double quest for execution speed and programming simplicity, modern hardware architectures include user-transparent performance enhancing features (e.g., pipelining, caching). The presence of these elements complicates WCET estimation. Limiting generality to sequential code running on single-cores and selecting moderately complex hardware allows the preservation of computational tractability, while the hardware micro-architecture is precisely modeled. This allows the computation of tight execution time bounds.

* This work was partially supported by EU COST Action IC1202 Timing Analysis at Code-Level (TACLe)

© Dumitru Potop-Butucaru and Isabelle Puaut;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 21–31

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

WCRT analysis emphasizes the system-level complexity, by taking into account aspects such as inter-task task communication/synchronization, and interaction with the environment. The objective here is usually to provide execution time bounds for execution flows involving several tasks, possibly running on multiple processors, and their communications and synchronizations. To limit computational complexity of WCRT analysis, hardware and software are usually represented in much less detail than in WCET estimation techniques. Typical objects at this level are sequential tasks characterized by functional and non-functional properties, such as: inputs and outputs, WCET, period, execution conditions, etc.

When dealing with parallel applications running on multicore architectures, the classical separation between WCET and WCRT analysis has to be revisited, since an application, even when considered in isolation from the others, includes parallelism. In this paper, we address the issue of determining the WCET of an isolated parallel application where each core is statically allocated one sequential thread. Since the threads synchronize with each other, our *integrated* WCET estimation technique must address issues that are usually dealt with by WCRT estimation techniques (integration of synchronization and communication costs). On the other hand, as a WCET estimation technique, our proposal calculates execution times of a parallel application considered in isolation from the other activities that run concurrently on the multi-core architecture.

Contribution. Classical WCRT-based timing analysis techniques for parallel code isolate micro-architecture analysis from the analysis of synchronizations between cores by performing them in two separate analysis phases (WCET and WCRT analysis). This isolation has its advantages, such as a reduction of the complexity of each analysis phase, and a separation of concerns that facilitates the development of analysis tools. But isolation also has a major drawback: a loss in precision which can be significant. To consider only one aspect, to be safe the WCET analysis of each synchronization-free sequential code region has to consider an undetermined micro-architecture state. This may result in overestimated WCETs, and consequently on pessimistic execution time bounds for the whole parallel application. The contribution of this paper is an *integrated* WCET analysis approach that considers at the same time micro-architectural information and the synchronizations between cores. This is achieved by extending a state-of-the-art WCET estimation technique and tool to manage synchronizations and communications between the sequential threads running on the different cores. The benefits of the proposed method are twofold. On the one hand, the micro-architectural state is not lost between synchronization-free code regions running on the same core, which results in tighter execution time estimates. On the other hand, only one tool is required for the temporal validation of the parallel application, which reduces the complexity of the timing validation toolchain.

Such a holistic approach is made possible by the use of deterministic and composable software and hardware architectures (homogeneous multi-cores without cache sharing, static assignment of the code regions on the cores) as detailed later in this paper. We demonstrate the interest of the approach using an adaptive differential pulse-code modulation (*adpcm*) encoder where the integrated WCET approach provides significantly tighter response time estimations than the more classical WCRT approaches.

Outline. The rest of this paper is organized as follows. Section 2 presents the application model and defines more formally what is meant by worst-case execution time of a parallel application. Section 3 details and motivates the class of multi-core architectures considered in this study. Section 4 defines our WCET estimation method. Experimental results are given in Section 5. Our proposal is briefly compared to related work in Section 6. Finally, we conclude and discuss future work in Section 7.

```
void core1() {
  int tqmf[24]; long xa, xb, el;
  int xin1, xin2, decis_levl;
  for(;;) { //Infinite loop
    // Computation phase 1
    xa = 0; xb = 0;
    for (i=0;i<12;i++) { // 12 iterations
      xa += (long) tqmf[2*i]*h[2*i];
      xb += (long) tqmf[2*i+1]*h[2*i+1];
    }
    // Send the results to core 2
    send(channel1,(int)((xa+xb)>>15)); ------
    // Read inputs
    xin1=read_input(); xin2=read_input();
    // Computation phase 2
    for(i=23;i>=2;i--) { // 22 iterations
      tqmf[i]=tqmf[i-2];
    }
    tqmf[1] = xin1; tqmf[0] = xin2;
    // Receive data from core2 and output it
    decis_levl = receive(channel2) ; <------
    write_output(decis_levl) ;
  }
}
```

```
const int decis_levl [30];
int core2() {
  int q,el;

  for(;;) {//Infinite loop






    // Receive data from core1
--> el = receive(channel1);
    // Computation phase 1
    el = (el>=0)?el:(-el);
    for (q = 0; q < 30; q++) {
      // 30 iterations
      if (el <= decis_levl[q])
        break;
    }
    // Send result to core1
--- send(channel2,decis_levl) ;

  }
}
```

Figure 1 Toy example: parallel version of *adpcm*, from the Mälardalen WCET benchmark suite [8].

2 Application model and problem formulation

The simplest embedded control systems running on *mono-processor* architectures follow a so-called *simple control loop* paradigm. In such systems, the software is simply a loop whose body is the sequence of calls to the various input sampling, processing, and actuation functions. The sequence of calls is fixed off-line. In this paper, we consider the multi-core equivalent of simple control loops, where each core executes a simple control loop (the practical importance of such a code structure mainly derives from the use of automatic mapping techniques [7], which often generate such code). We shall denote with task τ_i the program that forms the body of the loop executed by core $CPU_i, 1 \leq i \leq n$. Each task τ_i satisfies the classical requirements allowing WCET analysis (all loops it contains except the main loop have statically bounded numbers of iterations).

Tasks $\tau_i, 1 \leq i \leq n$ can communicate with each other through a set of logical message-passing channels $\mathcal{C} = \{c_1, \ldots, c_m\}$ which are bounded FIFO buffers that do not lose, duplicate, or corrupt messages. Communication is done using *send* and *receive* primitives that can be invoked at any statically known position in task τ_i. The two primitives are blocking (*send* on full channel, *receive* on empty channel), which means that channels can be used for synchronization. For the scope of this paper, we make the following assumptions concerning the channels: (i) each channel connects exactly two processors (one sender and one receiver); (ii) Each channel allows the storage of only one message. No assumption is made on how the logical message passing channels are implemented on the execution platform. We also assume that inter-task communications are free of deadlocks by construction.

An illustrating application, that will be used all along the paper, is given in Figure 1. The application is a portion of a bi-processor parallel version of *adpcm* (adaptive pulse code modulation) from the Mälardalen WCET benchmark suite [8]. We emphasized with arrows the two `send`/`receive` pairs associated with `channel1` and `channel2` respectively.

Assuming the previously-defined application model, the worst-case execution time analysis problem we solve in this paper is to compute the worst-case duration of a fixed number of iterations of the application, considered in isolation from the other activities running concurrently on the multi-core architecture.

3 Execution platform

As noted by Puschner *et al.* [14], obtaining *precise* and *composable* timing information in a multiprocessor system is only possible if we can ensure spatial or temporal separation between concurrent accesses to shared resources. The shared resources we consider in our work are the memory subsystem, the on-chip buses and networks (including I/O), DMA controllers, and the synchronization subsystem. We shall make on all of them hypotheses that allow both a precise timing characterization of complying architectures, and the modeling of complex, real-life architectures. Multi-processor systems with shared caches, although amenable to WCET estimation, may yield pessimistic WCET estimates, because the state of these caches becomes difficult to approximate in the presence of concurrent requests. Our choice is to consider architectures where each processor has its private cache subsystem, independent from the ones of other processors. It is also assumed that each core has *separate* instruction and data caches. We consider such architectures because separate caches are analyzable more precisely than unified caches by WCET estimation techniques. All caches have a Least Recently Used (LRU) replacement policy. LRU is selected because it was shown to be the most predictable cache replacement policy [15]. Finally, cores are homogeneous (all cores have the same micro-architecture).

Another significant source of WCET estimation imprecision is the presence of shared memory banks and shared communication busses. Our choice here is to consider architectures where the duration of all memory accesses and data transmissions can be precisely determined. The timing precision can be ensured: (i) fully by hardware mechanisms, for instance through the use of time division (TDM) memory controllers or on-chip buses [5], (ii) or through a mix of software and hardware mechanisms. In these cases, software and/or hardware synchronization mechanisms (semaphores, locks) are used to guarantee the absence of contentions due to access to RAM banks or communication buses. In this paper we consider architectures of the second type, as this case covers classical distributed bus-based architectures, shared memory architectures featuring multiple RAM banks, but also mixes of the two, such as the Network-on-Chip (NoC) based architectures proposed by various vendors [17, 12]. Our NoC-based experimentation platform that will be detailed in Section 5 falls in this last case. An upper bound of the communication latency for every *send*/*receive* pair is assumed known, which is realistic on all the previously-mentioned architectures. The determination of communications latencies is considered outside the scope of the paper.

4 WCET computation

Our approach to WCET computation for parallel applications (§ 4.2) consists in extending a state-of-the-art WCET estimation method (§ 4.1) to compute WCETs of parallel applications.

4.1 Existing state-of-the-art WCET estimation technique

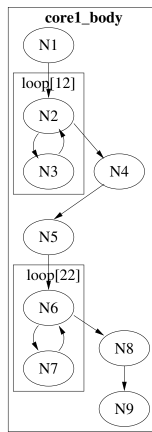
Static WCET estimation techniques are commonly organized in three phases performing different analyses [4]: *Control-flow analysis*, *Hardware-level analysis* and *WCET calculation*.

Control-flow analysis. This phase extracts information about possible execution paths from the program source or binary. The output of this phase is a data structure representing the possible flows. For the scope of this paper, this phase produces Control Flow Graphs (CFG), extracted from the program binary. The CFGs are annotated with additional flow

information such as maximum number of loop iterations. The CFG of the loop body of the task running on *core1* (from our sample application) is given in Fig. 2(a).

Hardware-level analysis. This step, also called *low-level analysis*, estimates the worst-case execution times of basic blocks. The difficulty during this phase is to take into account micro-architectural components of the target processor (caches, pipelines, branch predictors). In the presence of such components, the execution time of a statement is dependent on the context it is called in. The overall typical outcome of hardware-level analysis is a maximum execution time per basic block in two different contexts to cope with cache effects: the first execution of the basic block in a loop, denoted t^f and its subsequent executions, denoted t^n, as more formally defined in [9]; (negative) execution times may also be associated to edges to account for pipeline effects between basic blocks.

WCET calculation. The purpose of this final phase is to determine an estimate for the WCET, based on the flow and timing information derived in the previous phases. The most widespread calculation method, that will be adopted in this paper, is called *implicit-path enumeration* (IPET). In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. Each entity (basic block or program flow edge) in the code is assigned two values: a time coefficient, denoted t_{entity}, which expresses the upper bound of the contribution of that entity to the total execution time every time it is executed, and count variable (x_{entity}), corresponding to the number of times the entity is executed. The program's WCET of the program is determined by maximizing the sum of products of the execution counts and times ($\sum_{i \in entities} x_i * t_i$), where the execution count variables are subject to constraints reflecting the structure of the code and possible flows. The result of an IPET calculation is an upper timing bound and a worst-case count for each execution count variable.



(a) CFG of the loop body of `core1`

// Start constraint
$x_1 = 1$
// Structural constraints
$x_1 = x_{1,2}$
$x_2 = x_{1,2} + x_{2,3} = x_{2,3} + x_{2,4}$
$x_3 = x_{2,3} = x_{3,2}$
$x_4 = x_{2,4} = x_{4,5}, \ldots$
// Loop bound constraints
$x_3 \leq 12$
$x_7 \leq 22$
// Cache-induced constraints
$x_1 = x_1^f + x_1^n$
$x_1^f \leq 1, \ldots$
// WCET expression
$\text{maximize}(x_1^f * 10 + x_1^n * 10$
$+ x_2^f * 120 + x_2^n * 14 + \ldots$
$+ x_{1,2} * -1 + \ldots);$

(b) Constraints for WCET calculation of `core1_body`

Figure 2 State-of-the-art WCET calculation on the loop body of `core1` in our illustrating example.

Fig. 2(b) illustrates the constraints and formulas generated by an IPET-based bound calculation method on the loop body of `core1`, assuming it is the program entry point. The start constraint states that the code is executed once. The structural constraints reflect the possible program flows, meaning that each basic block must be entered the same number of times as it is exited. The loop bounds constrain the number of executions of basic blocks inside loops. Cache induced constraints express that basic blocks have different execution times, one for their first execution, another for the next ones. In the WCET expression to be maximized, there are two execution durations per basic block to model cache effects. For instance, in the formula, $t_2^f = 120$ (first execution, cold cache), whereas $t_2^n = 14$ (subsequent executions, warmed-up cache).

4.2 WCET computation of parallel applications

Starting from the WCET estimation method sketched above, we build our WCET estimation technique for parallel applications by performing a per-core hardware-level analysis and then adding new edges in the CFG to model synchronization/communications between code regions. The modified analyzer runs as follows, with the analysis phases presented in their invocation order:

1. The original **control flow analysis** extracts the CFG of the tasks to be run on the cores, from the application binary.
2. The **hardware-level analysis** runs *unmodified* on each task (control loop running on each core). During this step, each task is analyzed as if it was not communicating with the other tasks executed concurrently on the other cores.
3. A new step dubbed **modeling of communications**, is invoked. This step adds new edges between the control flow graphs of these tasks, the result being a single CFG. A new edge is added for each communication between code regions; it is associated with a duration to model its execution time (message transmission time for communications).
4. The **WCET computation** step is executed unmodified, even though the CFG corresponds to a parallel application and has slightly different topological properties. This is due to the fact that the analysis works by finding the critical path in a directed acyclic graph. The fact that the graph represents purely sequential behaviors, or parallel ones (including the new edges that model communications) is not important.

The method was integrated into the Heptane static WCET estimation tool [1] through the addition of a new pass, corresponding to phase 3, interposed between hardware-level analysis and WCET calculation. Communications between code regions are detected through annotations in the source code of the analyzed application, specifying at each communication point the recipient of the message and the communication latency. The code of the new pass represents around 200 lines of C++ code.

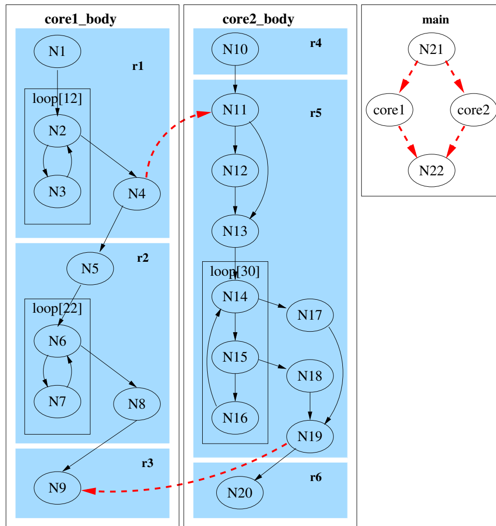


Figure 3 WCET computation of parallel application.

The method is illustrated step by step in Fig. 3 on our toy application of Fig. 1. The shaded areas labeled **r1**–**r6** correspond to the code regions, which are by definition the portions of the two tasks that are separated by communications. For instance, node N4 of the `core1_body` CFG is the basic block containing the `send` call on `channel1`. After the hardware-level analysis phase runs unmodified on the tasks of the parallel program, the modeling of communications adds new edges in the application CFG to model communications (bold arrows in the figure). These new edges correspond to: message passing between code regions (edges $N4 \rightarrow N11$ and $N19 \rightarrow N9$) and parallel launching of code regions on the different cores (edges to and from nodes $N21$ and $N22$ in the application entry point *main*).

During the hardware-level analysis phase, our WCET analysis method applies instruction cache, data cache, and pipeline analysis on the two CFGs `core1_body` and `core2_body`. This allows to benefit from the tightness of hardware-level analysis on each task. For instance, in task `core1_body`, it allows to detect that array *tqmf* is still in the data cache after calling primitive *send*. This would not have been possible if a *decoupled* approach was used (WCET estimation of regions followed by an aggregation of individual WCETs to compute the global WCET). If a decoupled method was used, conservative assumptions would have been taken for the analysis safety (assuming the worst-case hardware state, i.e. empty cache at WCET analysis start). Using an integrated approach, the hardware-level analysis is able to capture hardware effects between regions (instruction caches, data caches, pipeline) naturally.

Finally, the WCET computation step is applied unmodified. Thanks to the introduction of the new edges, new constraints are automatically added in the WCET calculation equations, and communication delays are automatically taken into account. The new or modified formulas of the WCET calculation equations are illustrated below for our running example, with the modified parts in bold face. Communication/synchronization edges are taken into account in the new structural constraints (e.g. number of executions of communication/synchronization edges, $x_{4,11}$). Data transmission latencies are considered as well (e.g. 250 time units to communicate data from node $N4$ to node $N11$ according to the amount of data to be transmitted between the two nodes).

// New or modified structural constraints (non exhaustive)
$x_4 = x_{2,4} = x_{4,5} + \mathbf{x_{4,11}}$
$x_{11} = x_{10,11} + \mathbf{x_{4,11}} = x_{11,12} + x_{11,13}$
// New WCET expression
$\text{maximize}(x_1^f * 10 + x_1^n * 10 + x_2^f * 120 + x_2^n * 14 + \mathbf{x_{4,11}} * \mathbf{250} + x_{1,2} * -1 + ...);$

5 Experimental evaluation

5.1 Experimental setup

Multi-core architecture. Given that our claims mainly concern the precision of the timing analysis, we considered an evaluation platform allowing us to perform cycle-accurate estimations and measurements of execution time, in both single-processor and multi-processor cases. We achieved this by using the SoCLib library [16] for virtual prototyping of multi-processor systems-on-chips (MPSoC). The hardware components we use are of cycle-accurate, bit accurate type, written in SystemC.

The precise architecture we worked on using SoCLib is a scaled-down version of that of [2]. While the original platform scales up to 4096 cores, we have only used for the presented experiments single-, double-, and quad-core configurations. Each core has separate L1 instruction and data caches, both implementing a Least Recently Used (LRU) cache replacement policy. Both caches feature 32 sets, 4 ways, and 32 bytes per cache line. All CPU cores are of the same type, using the MIPS32 instruction set. Each core is part of a computing tile containing a multi-bank RAM (to accommodate non-interferent concurrent accesses to program text and data by the CPU cores), a DMA unit, and a hardware lock unit. The local interconnect of each tile is a full crossbar. The tiles are inter-connected through a 2D mesh network-on-chip. The overall structure of our architecture is very similar to that of commercial many-core architectures [12].

Studied applications. The proposed WCET estimation method was experimented on two small signal processing applications. The first one is a parallel version of the *adaptive differential pulse-code modulation (adpcm)* from the Mälardalen WCET benchmark suite [8]. The global dataflow of the code executed at each iteration of the modulation application is depicted in Fig. 4, where boxes represent code regions and arrows communications between them. Fig. 4 also depicts mapping of regions to cores when the application is parallelized for a 2-core architecture. Only the arrows crossing CPU boundaries are coded as communications; the sequencing of QMF, low-band encoder and multiplexer is implemented simply by calling successively the three codes in the main loop of CPU0. When parallelized for a 4-core architecture, every region is assigned to a different core, and software pipelining is used to allow more parallelism. The communication latency of every inter-core communication was determined by an analysis of the hardware platform as a formula dependent on the volume of data to be transferred.

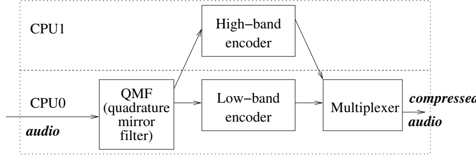


Figure 4 *adpcm* application: dataflow and mapping on a 2-core architecture.

The second application, named *filter*, is a simple load balancing example, where two processors are needed to improve the throughput of a simple image filter. In this bi-processor application, processor 0 successively receives image lines in a buffer. The buffer content must be stored elsewhere to allow a new line to arrive, and this new line will be sent to processor 1, cyclically.

Application code was compiled using a standard GNU MIPS compilation toolchain with no optimization. For the scope of this performance evaluation, application code was parallelized manually. Automatic code parallelization software like [6] or offline real-time scheduling tools like [7] that generate efficient parallel code could have been used instead.

5.2 Experimental results

Experimental results are given: (i) to evaluate the accuracy of the hardware model used in the base timing analysis tool; (ii) to compare WCETs obtained using our *integrated* approach against those obtained using a *decoupled* estimation method; (iii) to evaluate the pessimism of our integrated method. We do not give numbers on the run-time of the analysis, simply because modifying the application's CFG turned out to take negligible time compared to hardware-level analysis and WCET calculation.

Accuracy of hardware model. To show the accuracy of the hardware model used in the analysis, we have validated Heptane's hardware model against the SoCLib simulator on single-path code. Experiments were conducted on randomly generated single-path code, starting with known contents of the instruction and data caches. After a careful and extensive comparison of the analyzer and simulator cycle counts, both tools returned exactly the same number of cycles for all considered code.

Comparison with baseline decoupled WCET estimation method. To evaluate the tightness of WCET estimates, we have compared them with a baseline decoupled approach that first estimates WCETs of code regions and then computes the overall WCET through a composition of the regions WCETs. The baseline method operates as follows. It first computes computes WCETs of all regions; to be safe, the worst-case hardware state is assumed by the

Table 1 Experimental results: computed WCET bounds using our integrated approach and a base-line decoupled approach, and measured execution time. Improvement over the baseline is defined as $\frac{Decoupled-Integrated}{Integrated} * 100$. Analysis pessimism is defined as $\frac{Integrated-Measured}{Measured} * 100$.

Name	Integrated	Decoupled	Gain (%)	Measured	Pessimism (%)
adpcm – 2 cores	73563	101431	**36.5%**	64944	**13.3%**
adpcm – 4 cores	44568	55919	**25.5%**	41468	**7.5%**
filter – 2 cores	110825	112543	**1.55%**	108296	**2.3%**

static analyzer at the start of every region. Then, the application overall WCET is computed in an *ad hoc* manner according to the synchronization pattern between code regions. This turned out to be very easy for the considered applications, that have simple and regular communications, never more complex that the ones illustrated in Figure 1.

The estimated WCETs are given in Table 1, for 10 iterations of the main control loop on each core. The WCETs produced by our integrated approach are always tighter than using the decoupled method (21% in average on the three case studies). The gain varies depending on the amount of reuse between successive regions assigned to the same core. When the amount of reuse is high, like in application *adpcm*, that features intensive code and data reuse between code regions, the gain is significant. When the amount of reuse is smaller like in *filter* (no reuse of data, modest reuse of code between regions), the gain is much smaller.

Comparison with observed execution times. The pessimism of our WCET evaluation method is evaluated by comparing estimated WCETs with observed execution times, obtained using the SoCLib simulation software. Regarding simulation results, due to time constraints, we made no attempt to identify the worst-case input data and execute the code with typical input data, not necessarily representative of the worst-case situation. The estimated pessimism is thus an upper bound of the method pessimism. Results are reported in Table 1. The numbers show that even without executing the code using its worst-case input data, the results are encouraging: estimated and measured execution times are close to each other (of 7.7% in average). Further experiments need to be conducted to identify the actual overestimation and not only an upper bound of the overestimation.

6 Related work

Much research effort has been spent in the past in estimating WCETs of sequential code and WCRTs of multi-task applications. Research on WCET estimation has mainly targeted software running on single-core architectures (see [4] for a survey). A lot of effort has been put on *hardware-level* analysis, allowing architectures with caches and in-order pipelines to be analyzed precisely. The research presented in this paper is *not* a new WCET estimation technique, but rather takes benefit of state-of-the-art low level analysis to produce tight WCET estimates of parallel applications.

Many WCRT estimation methods compute end-to-end response times of distributed applications communicating using message passing, or multiprocessor systems (e.g. [11]). To our best knowledge, all these methods can be qualified as *decoupled*, in the sense that they use as input WCET estimations of code regions computed before the WCRT analysis. By comparison, we have shown that an *integrated* analysis allows to produce tighter WCRTs than a decoupled approach, because it allows hardware effects between code regions to be captured accurately.

The research we found to be closer to our approach is described in [3, 13, 10]. Paper [3] is devoted to WCET estimation of a parallel application running on a predictable multi-core architecture. Similarly to our work, emphasis is put on predictability of the hardware and software architectures. However, in contrast to [3] that provides formulas to combine WCETs of code snippets to obtain the WCET of the parallel application, in our work the application running on each core is analyzed as a whole. As a consequence, we are able to exploit knowledge of the hardware state between code snippets and thus can provide tighter estimates, especially for fine-grain parallelism.

In [13], a method to determine residual cache states after the execution of sequential code on a mono-core platform is provided. The method allows to obtain tighter WCETs in case of repetitive executions of the analyzed code. Using our method, we obtain the same benefits, but without needing a specific analysis and tool. This benefit comes as a side product of our method because the WCET computation of the parallel application is integrated into a WCET estimation tool that originally was analyzing sequential code.

Paper [10] proposes an ILP formulation for WCRT computation of task graphs running on multi-core systems. The method computes the application WCRT given a task-to-core mapping, architecture and scheduling policy, with contentions when accessing shared resources. Unlike [10], we currently rule out resource contentions, that is left for future work. However, contrary to [10], our analysis of the hardware is expected to be tighter because of our *integrated* approach.

7 Conclusion

We have presented in this paper a method to compute the WCETs of parallel applications running on multicore platforms. Thanks to small modifications of a WCET computation method, the parallel application can be analyzed as a whole, such that hardware effects across code regions of the application are dealt with naturally. We have demonstrated that our approach produces WCETs that are tighter than using a classical method by 21% in average. Preliminary experiments show that the WCET over-approximation is below 7.7% in average. We believe that our method can be integrated easily in other WCET estimation tools using the implicit path enumeration techniques to the extent that the analysis framework is sufficiently modular (hardware-level analysis and WCET computation are clearly separated). The proximity of our test architecture to existing commercial many-core architectures also suggests that our results are easily transposable to them.

In this paper, assumptions have been made regarding the software structure in order to demonstrate the validity of our approach on simple but yet realistic setting. In future work, our first objective will be to relax as much as possible these assumptions to broaden the scope of application of the approach. Another area for future research will be to use obtained WCETs to refine the structure of the parallel application (mapping of code regions on the cores, execution order). Finally, scalability to a larger number of cores is another area for future work.

References

1 A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *ECRTS*, pages 37 –44, July 2001.

2 M. Djemal, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang. Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *DASIP*, 2012.

3 C. Rochange et al. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In *WCET workshop*, 2010.
4 R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7(3):36:1–36:53, May 2008.
5 K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.
6 M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S.P. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X*, 2002.
7 T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives. In *MEMOCODE*, 2003.
8 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET workshop*, 2010.
9 D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.
10 J. Kim, H. Oh, H. Ha, S. Kang, J. Choi, and S. Ha. An ILP-based worst-case performance analysis technique for distributed real-time embedded systems. In *RTSS*, 2012.
11 M. Kuo, R. Sinha, and P. Roop. Efficient WCRT analysis of synchronous programs using reachability. In *Proceedings DAC'11*, San Diego, CA, USA, 2011.
12 The MPPA256 many-core architecture. Online `http://www.kalray.eu/products/mppa-manycore/mppa-256/`, 2012.
13 F. Nemer, H. Cassé, P. Sainrat, and J.P. Bahsoun. Inter-task WCET computation for a-way instruction caches. In *SIES*, 2008.
14 P. Puschner, R. Kirner, and R. Pettit. Towards composable timing for real-time programs. In *STFSSD'09*, 2009.
15 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *RTSJ*, 37(2), 2007.
16 SoCLib: an open platform for virtual prototyping of multi-processors system on chip, 2011. Online at: http://www.soclib.fr.
17 The TilePro64 many-core architecture. Online `http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf`, 2008.

Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective*

Mihail Asavoae, Claire Maiza, and Pascal Raymond

Laboratoire Verimag
Centre Equation, 2 Avenue de Vignate, Gieres, France
{Mihail.Asavoae,Claire.Maiza,Pascal.Raymond}@imag.fr

Abstract

Advanced design techniques of safety-critical applications use specialized development model-based methods. Under this setting, the application exists at several levels of description, as the result of a sequence of transformations. On the positive side, the application is developed in a systematic way, while on the negative side, its high-level semantics may be obfuscated when represented at the lower levels. The application should provide certain functional and non-functional guarantees. When the application is a hard real-time program, such guarantees could be deadlines, thus making the computation of worst-case execution time (WCET) bounds mandatory. This paper overviews, in the context of WCET analysis, what are the existing techniques to extract, express and exploit the program semantics along the model-based development workflow.

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

Keywords and phrases Survey, WCET analysis, Program semantics, Model-based design, Infeasible paths

Digital Object Identifier 10.4230/OASIcs.WCET.2013.32

1 Introduction

Programming embedded and hard real-time systems requires careful considerations not only with respect to correctness criteria of the software product, but also to resource utilization (i.e. memory usage, power consumption or timing behavior). This implies to build the embedded and real-time applications in a systematic way and thus, to set the grounds for subsequent development of analysis tools. The worst-case execution time (WCET) analysis provides safe guarantees w.r.t. the timing behavior of hard real-time applications.

A popular solution in the direction of software systematization is called *model-based design* (MBD) and presents, in general, three components: a high-level specification language to develop the application (which is also called *model*); compilation support to further process the model; tool support for simulation (and in some cases analysis) purposes. We restrict our discussion on a particular MBD workflow which is currently used in both avionics and automotive domains and where the compilation support generates classical imperative code and then binary code as shown in Figure 1(C). In this setting, the application semantics is present in the high-level model, in the intermediate program and in the binary code. Apart from the semantics representation, the model-based workflow also has two semantics transfer levels: from model to imperative code and from imperative code to binary code. We discuss

* This work was partially supported by ANR under grant ANR-12-INSE-0001 and by EU COST Action IC1202:Timing Analysis On Code-Level (TACLe).

© Mihail Asavoae, Claire Maiza, and Pascal Raymond;
licensed under Creative Commons License CC-BY

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 32–41

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

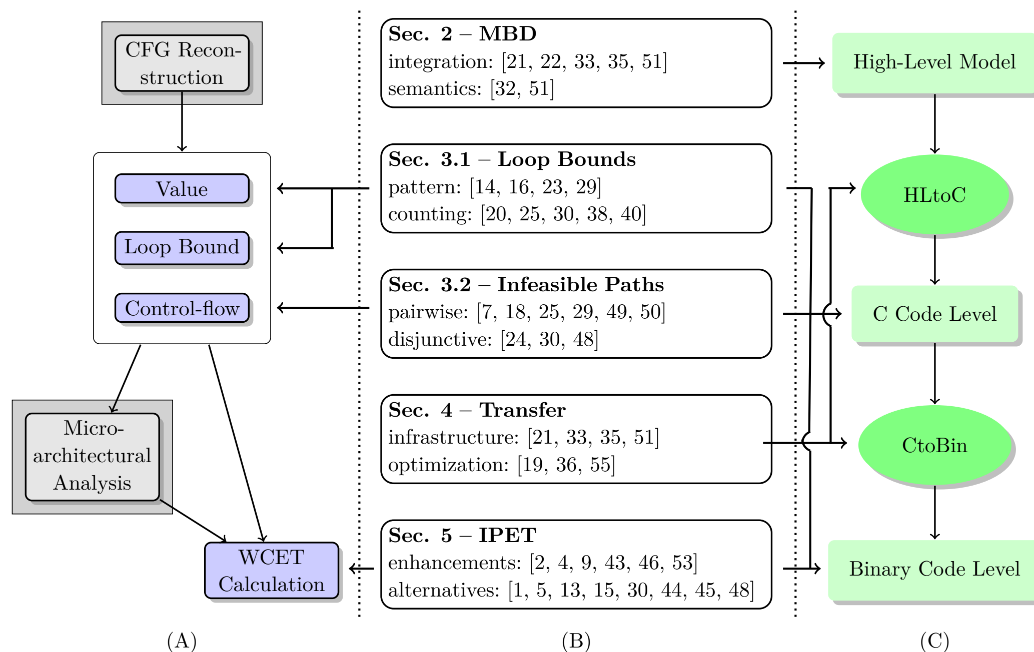


Figure 1 Static WCET analysis workflow (A), paper organization (B) and MBD workflow (C).

how the program semantics is expressed, extracted and exploited in such a workflow, when the analysis of interest computes WCET bounds.

The WCET analysis of a particular program is performed at the binary level and with knowledge about the underlying architecture. As it is summarized in [56] and shown in Figure 1(A), a typical workflow for a WCET analysis proceeds with the CFG extraction, a number of program flow- and processor-behavior analyses, and finally, the bound computation. The WCET analysis should provide safe and tight estimations of the actual WCET of a program. To address these, the WCET analysis workflow relies on a number of specific analyses, spawn from both the flow analysis (i.e. detection of loop bounds and infeasible paths) and the architecture analysis (i.e. cache and pipeline behavior prediction).

In this paper we present a survey study on how the WCET analysis workflow is projected on the MBD workflow, from a particular point of view – the separation of concerns at the level of program semantics manipulation. Due to the generality of the MBD framework and the multitude of contributions in the WCET analysis, as well as the current space limit, we restrict our presentation under a setting defined by the following constraints. First, we consider MBDs where the model is compiled into C code. Second, we consider the architecture-related analyses to be orthogonal to our investigation on the program semantics and thus are left out, as shown in Figure 1(A). This second restriction activates other intended omissions, from our survey: analyses for CFG extraction and for classification of load/store instructions. Third, we discuss the path-analysis problem from the popular implicit path-enumeration technique (IPET) point of view, classifying the approaches as enhancements or alternatives to it.

The works in [37, 56] survey, from certain angles, the state-of-the-art approaches in the WCET analysis field of research. The authors of [37] rely on the notion of the flow fact and classify then-existing WCET analysis approaches w.r.t. this notion. As a consequence, this allows comparisons between various approaches at the confluence of axis for the representation levels and the execution-time modeling. The survey in [56] is ampler and newer than [37],

covering both the methods and the existing tools in the WCET analysis field of research. The methods are classified into static and measurement-based ones, with the information presented at the level of WCET analysis subtasks. What we propose here is a specialized view, in the form of a separation of concerns from the program semantics perspective. In comparison, this paper is different because it covers (1) an up-to-date specialization of the representation levels axis from [37] (i.e. only the semantics levels in the model-based development frameworks with generation of C code capabilities) (2) an up-to-date specialization to the static-based methods from [56], in particular to the flow analyses, specialization which is presented on (3) a projection of the workflow of the model-based development frameworks. To summarize, our survey follows the organization of a modern software development framework for embedded and real-time applications and presents up-to-date works, exclusively from a program semantics perspective.

The organization of this paper follows the projection of WCET analysis workflow over the levels of the model-based development frameworks – Figure 1(B). We present in Section 2 the model-based frameworks as well as the current approaches towards WCET analysis on the general setting. In Section 3, we project the flow analysis of interest (i.e. loop-bound and infeasible path detection) at each programming language level in the development frameworks, while in Section 4 we discuss how to transfer information. We dedicate Section 5 to the path analysis problem, then we draw conclusions and discuss open problems, in Section 6.

2 Model-Based Development Framework

The development of embedded real-time applications using MBDs [11] gained popularity in the last decade. The key element lays in the design environment – using a high-level specification language with mathematical background and graphical support, which enables rapid prototyping and a high level of design reusability. Moreover, a MBD tool provides controller analysis and synthesis, as well as deployment support. Application development in the automotive [10, 47], avionics [54] and aerospace [31] domains rely on popular tools like Scade Suite and Matlab Simulink/Stateflow. More precisely, Scade belongs to the synchronous languages family, which means it was designed to generate code, while Matlab served initally as a simulation tool, but it is now equipped with code generation facilities.

The synchronous paradigm is a deterministic parallel programming style, which compiles synchronous programs into classical sequential C code with bounded loops and memory usage – and for which it is mandatory to find a WCET bound. Synchronous programming languages can be classified into data-flow oriented (e.g. Lustre), control-flow oriented (e.g. Esterel) and mixed approaches (e.g. Scade Suite). The Lustre programming language and its formal semantics are presented in [26]. Semantically, a Lustre program transforms input streams of values into output streams of values and structurally, it represents a system of equations defining the variables (these are functions from time domain to value domain). The Esterel programming language and its formal semantics are presented in [6]. Semantically, an Esterel program reacts to input events (i.e. signals) by producing output signals, and structurally, it consists of specialized imperative statements to specify control operations (i.e. delay, signal emission, abortion etc). Both the Lustre and Esterel compilers [27, 17] generate sequential C code from the intrinsically-parallel synchronous program. The mixed approaches, represented by Scade Suite and Matlab Simulink/Stateflow provide powerful modeling languages to integrate data and control-related aspects. For data- and control-flow parts of the application, Scade uses Lustre and respectively Safe State Machines (graphical equivalent to Esterel), while Matlab uses the languages Simulink and respectively Stateflow.

The safety-critical applications, which are developed through MBDs, often require guarantees about their timing behavior. Particularly, the idealized "instantaneous" synchronous tick is implemented as a guarantee of an upper bound on the execution time. Therefore, it is required to integrate the WCET analysis techniques into the MBD workflow. There, we state the following two types of contributions: *integration methods* (w.r.t. the timing analyzer) and *semantics-specific methods* (w.r.t. the model or other program representation-level).

Integration methods. The integration methods simply embed the timing analyzer in the MBD workflow as tool support in the application development process. Timing analyzers are integrated in Scade Suite workflow [21] (using the existing traceability properties of the MBD), in Matlab Simulink [35, 51], in an Esterel-driven MBD [33] and in an automotive-specific model-based development framework, called Ascet [22].

Semantics-specific methods. The semantics-specific methods focus on the precision of the WCET analysis, transporting program semantics properties from the model level to the binary level. Existing approaches investigate the timing behavior of Matlab Simulink/Stateflow models [51] as well as of Esterel applications (during one tick and along multiple ticks) [32]. The program semantics information materializes into various path pattern types [32] or entailment relations and flow constraints [51]. The works in [8, 3] propose standalone timing analyses of synchronous programs, without a complete integration into the WCET analysis workflow. Both perform timing analyses of Esterel code which is executed during one tick, and also called worst case reaction time (WCRT) analyses.

3 Representation Level – Language

The model-based design presents several levels of program representation. In this paper, we consider those design platforms with C code as their representation language between the model and the binary level. For the WCET analysis techniques, it is (1) convenient to work on the low-level representation because of the architecture-related information, and (2) inconvenient because of the obfuscated program structure (and possibly the semantics), due to compilation influence. The MBD frameworks open the possibility to manipulate the program semantics at a convenient level. There are two complementary methods to obtain the program semantics properties of interest: add *manual annotations* or extract them using *dedicated analyses*. We briefly cover the former and then, elaborate on the latter.

Manual methods. In general, the MBD workflow provides annotation support through intermediate generated files (e.g. for Scade Suite models [51]) and it handles information about code locality. Nevertheless, it is possible to extend the given MBD workflow to accommodate specific WCET analysis annotations (e.g. for Matlab Simulink/Stateflow models [35]). For a more general view on existing annotation languages and tool support in the WCET analysis domain, we recommend the comprehensive survey in [34].

Automated methods. We focus on the following two subtasks performed on the CFG representation of the program, the loop bound analysis and the control-flow analysis (with the infeasible path detection). Most of the existing solutions work at the binary level.

3.1 Loop Bounds Detection

The model-based design framework should generate certified C code, which implies it to be deterministic and traceable, without dynamic allocation, with checked dynamic accesses etc.

As a result, such C code contains only bounded loops, usually in the form of for-statements. However, complex models could produce preemptive conditions to break out of the loops, making the initial loop bound a grossly overestimation of the actual number of iterations. The WCET analysis requires knowledge of both loop and recursion bounds. Therefore, a loop bound analysis attempts to automatically infer such bounds, or in other words to discover inductive invariants over loop counters. The general procedure consists of three steps: express how loop variables change across iterations, solve the resulting expressions (i.e. obtain their closed form) and finally, project the results over the loop counters [41]. Because this general procedure is undecidable, the state-of-the-art approaches rely on pattern-based heuristics at the level of loop structure and/or loop data. We classify the loop bound analyses w.r.t. whether they employ *pattern-matching* or *counting*, and further projected on how the loop bound is expressed.

Pattern-based methods. We consider the loop bound analyses which use patterns at the level of code constructs [29, 16, 23] and/or encode the closed forms expressions of interest [16]. The code-centered analyses considers specific patterns of loop tests (e.g. comparisons of variables to constant values) and captures how loop variables change across iterations through data-flow analysis [29, 23, 14] or abstract interpretation [16]. The property-centered analyses uses pattern matching in a different way: the results of an abstract interpretation-based analysis match user-defined patterns representing closed forms expressions [16]. The results of these loop bound analyses are expressed as summations [29], intervals [23, 14] or both [16].

Counting-based methods. We consider loop bound analyses which symbolically accumulates knowledge about (i.e. count) the number of loop iterations. The key element of such an analysis is the loop counter – a program or an analysis-specific variable. More specifically, the loop counter can be: a symbolic variable with an interval domain [25, 40], a number of program states (modulo equivalence classes) [20], a Presburger set-representation of a symbolic variable [30], a parameter of a recurrence equation [38] or a formula [12]. The counting methods are: derivations from abstract interpretation (i.e. abstract execution) [25], combinations of abstract interpretation with other methods (i.e. program slicing) [40, 20] or SMT-based invariant generators [38]. The results of these loop bound analyses are expressed as summations [38], intervals [25, 40, 20] and those of [30] could be disjoint sets of values (i.e. specific bounds or intervals).

3.2 Infeasible Paths Detection

The ability to detect infeasible execution paths greatly influences the precision of a WCET analysis. The sequence of instructions which define a program execution may be characterized by the sequence of decisions taken at conditional statements. A program execution is infeasible when it cannot be exercised, regardless of the input data. A common way to identify the infeasibility is to detect conflicting pairs of conditional statements. Note that, in the context of WCET analysis, the infeasibility expressed as conflicting pairs hides a more practical aspect – it could be easily encoded in the popular IPET formulation of the path analysis. Nevertheless, there exists a conceptually orthogonal approach which is capable to detect more expressive transition (e.g. disjunctive) invariants. The application development through MBDs produces code with infeasible paths, coming from multiple sources: the model semantics, the specificities of the high-level language or the code generation techniques. For example, the control-flow aspects (e.g. specialized instructions or finite state machines) of the high-level language are translated into conflicting tests (to isolate impossible behaviors). Also, the underlying scheduling mechanism to generate deterministic C code could produce

repeating tests. Next, we classify the infeasible path detection analyses w.r.t. the result type: *conflict-pairs invariants* and *transition (e.g. disjunctive) invariants.*

Conflict-pair invariants methods. The methods to detect conflicting conditional statements are, in general, based on abstract interpretation methods [29, 18, 25, 7, 49], but search-based techniques are possible [50]. The general workflow has a value analysis phase, which defines possible values for program variables, and an extraction phase to produce relations between program statements (i.e. test-test or assignment-test). A number of specialized techniques aims at increasing the precision of the analyses: program slicing techniques [25, 49] or symbolic propagation [7]. The techniques in [25, 29, 50] discover infeasible paths in the context of the IPET technique. Moreover, these analyses are performed at the binary level, in the MBD hierarchy.

Disjunctive invariants methods. The methods to discover more infeasible paths use, either an expressive flow facts encoding (i.e. Presburger sets) [30], or techniques to expose the infeasibility [48, 24], via unfolding the set of program paths. This set is represented as a graph [48] (and explored with path pruning and graph refining techniques) or as a regular expression [24] (and explored with static analysis techniques). All these approaches [48, 24, 30] discover disjunctive invariants, which w.r.t. the dominant IPET technique (in the WCET analysis community), are difficult to express/exploit.

4 Representation Level – Transfer

We investigate how a particular MBD workflow with two levels of information transfer integrates the WCET analysis workflow. In general, transferring timing specific information (i.e. annotated or computed flow facts) from the high-level language to the imperative code level is well studied. The MBDs are supported by compilers which generate C code in a systematic way [27, 17, 52] and therefore offer good traceability information (e.g. the *KCG* compiler of Scade [52]). However, the second transfer level, from C to the binary level is more tricky because the general-purpose compilers such as *gcc*, feature code and data optimizations which affect the traceability. There are several classifications of traceability: depending on the direction of semantics transfer (i.e. forward and backward), modifications of the underlying tool support (i.e. deep and surface) or the presence of compiler optimizations [55]. We elaborate next on the classification based on the required infrastructure modification, then we overview existing approaches for traceability through compiler optimizations.

Infrastructure-based classification. Several approaches, e.g for Scade Suite [21, 51] rely on the available traceability information to integrate a timing analyzer into the MBD workflow. From an implementation point of view, traceability through annotations does not require modifications of the underlying structures. The Scade workflow uses XML files to transfer program location-based annotations. MBDs like Matlab Simulink [35] or Esterel [33] use modified infrastructure to improve the existing traceability. This type of traceability is achieved through the code structure and addresses the needs to transfer scope-based flow facts (e.g. loop bounds). Another application is to reconstruct the longest path returned by the timing analyzer, building a forward traceability chain as annotated ASTs [33].

Optimization-based classification. A more difficult problem is to transfer the flow facts for the WCET analysis, through compiler optimizations. The general strategy is to identify classes of optimizations and to model, case by case, the code transformations implied by the optimizations. The existing approaches [19, 36] require specialized languages to express flow

facts and their transformation. To integrate these languages, the compiler is either directly modified [19] or wrapped and manipulated by additional software infrastructure [36].

5 Path Analysis

The WCET analysis produces the timing bound (i.e. the longest execution path) after a path analysis phase. The longest path search implies that all the execution paths should be considered and it requires an underlying semantics model of them. We enumerate: control flow graph [39], abstract syntax tree [13], Kripke structure [42], timed automaton [15]. Nevertheless, an implicit path enumeration technique (IPET) formulation of the path analysis is the most popular approach for WCET analysis. We classify the path analyses into IPET-based algorithms (including enhancements of the original technique) and alternative approaches (syntax-directed schema, model checking or graph transformation).

IPET. The control flow graph (CFG) captures, in a compact way, the flow of the particular program, abstracting away data aspects. While there are several ways to represent the CFG, the WCET analysis considers the nodes as basic blocks (single-entry single-exit sequences of statements). The basic block representation of the CFG is used to encode the program paths as an ILP problem, and perform path analysis as ILP solving. The approach is called the implicit path enumeration technique (IPET) [39]. The ILP problem consists of two kinds of constraints: structural (or flow) constraints, to express input-output flow relations for basic blocks, and functional constraints, to handle loops (i.e. given as loop bounds) and to improve the precision (i.e. encode infeasible path). Specialized techniques extract ILP constraints from the CFG [18] or from other graph-based representations of the program [46, 53].

IPET enhancements. The overall method for path analysis through ILP solving suffers from two drawbacks: the timing bounds are as precise as the quality of the functional constraints and the size of the ILP problem directly affects the computation time of the results. Solutions for the former are presented in Section 3; next we focus on existing techniques to solve the ILP problem more efficiently. There are three complementary techniques: modular solving [4], problem size reducing [43] and parametric analysis [2, 9]. A modular solution identifies ILP sub-problems as CFG regions with single-entry single-exit properties, for which the locally computed results replaces the region. The size-reducing solution uses CFG transformations to combine conditionals and to reduce the number of program paths. The parametric solution produces, after specific analyses (for parametric dependencies between program variables and parametric expressions for loop bounds etc), a symbolic ILP problem.

IPET alternatives. Different path analysis methods project the representation of all program paths on syntactic and semantics artifacts. We enumerate the following solutions: syntax-directed [13, 44, 45], path-based [48], state-based [42, 15], graph-based [1] and special annotations [5, 30]. A syntax-directed approach uses timing schema for programming constructs and the path exploration is the AST traversal. A path-based WCET computation searches for the longest path among the previously computed bounds for different program paths. A model-based approach uses representations for the program states and the path exploration is performed with model checking techniques. A graph-based approach applies graph algorithmics to transform and/or traverse the CFG of the program. The annotation-based technique, which bridge the gap between the program semantics and the timing model [5, 30], transforms the path analysis into (one or more) constraint solving problems. With respect to the MBD, the path analysis is represented outside the MBD workflow, as an external procedure.

6 Concluding Discussions

This paper offers a broad view on a series of techniques for WCET analysis, with an emphasis on how the program semantics is manipulated. Moreover, this paper advocates for a separation of concerns at the level of program semantics – i.e. the separation into extract, express and exploit phases. We distinguish the following two directions of interest. First, in the context of the MBD workflow, it is necessary to manipulate the program semantics at various levels in the design chain. As such, the WCET analyzers would not only be integrated into MBDs [21, 35], but produce tighter results [51, 33], based on the model semantics. Second, the IPET technique dominates many approaches for WCET analysis, and, as such, many specializations of WCET analysis subtasks exist. The detection of loop bounds [16, 28] or infeasible paths [18, 29, 50] generate flow facts which are directly expressible into integer linear programming. As such, WCET analysis benefits from alternative approaches on flow facts generation [48, 24] or path analysis [30, 1].

Design and implement applications for embedded and hard real-time systems have several benefits when using a model-based design environment. First, the high-level language permits modular development, has formal semantics and capability for imperative code generation. From a WCET analysis perspective, (1) it allows annotations at the design level as opposed to cumbersome instrumentation at the low-level and (2) the resulting imperative code features good traceability because it adheres to certain certification criteria. Second, because the imperative code is systematically constructed, it opens new possibilities to apply specific discovery, express and transfer the flow properties to the WCET analysis level – the binary code. Third, the gap between the high-level driven MBD workflow and the low-level driven WCET analysis workflow requires a bi-directional transfer of the program semantics.

References

1. E. Althaus, S. Altmeyer, and R. Naujoks. Precise and efficient parametric path analysis. In *LCTES*, 2011.
2. S. Altmeyer, C. Humbert, B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *RTCSA*, pages 367–376, 2008.
3. S. Andalam, P. Roop, and A. Girault. Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs. In *DATE*, 2011.
4. C. Ballabriga and H. Cassé. Improving the wcet computation time by ipet using control flow graph partitioning. In *WCET*, 2008.
5. G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *RTP*, 2000.
6. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program. (SCP)*, 19(2):87–152, 1992.
7. R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI*, pages 146–158, 1997.
8. M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *ENTCS*, 203(4):65–79, June 2008.
9. S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. *Journal of Systems Architecture*, pages 614–624, 2011.
10. S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static wcet analysis to automotive communication software. In *ECRTS*, pages 249–258, 2005.
11. P. Caspi, P. Raymond, and S. Tripakis. Synchronous languages. In *Handbook of Real-Time And Embedded Systems*. Chapman and Hall, 2007.

12 J. Coffman, C. Healy, F. Mueller, and D. Whalley. Generalizing parametric timing analysis. In *LCTES*, pages 152–154, 2007.

13 A. Colin and I. Puaut. A modular & retargetable framework for tree-based wcet analysis. In *ECRTS*, pages 37–44, 2001.

14 C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *WCET*, 2007.

15 A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *WCET*, pages 113–123, 2010.

16 M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA*, 2008.

17 S. A. Edwards. Compiling esterel into sequential code. In *DAC*, pages 322–327, 2000.

18 J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, pages 163–174, 2000.

19 J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution times analysis for optimized code. In *ECRTS*, 1998.

20 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

21 C. Ferdinand, R. Heckmann, T.L. Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *ERTS2*, 2008.

22 C. Ferdinand, R. Heckmann, H.-J. Wolff, C. Renz, O. Parshin, and R. Wilhelm. Towards model-driven development of hard real-time systems. In *AASSD*, 2006.

23 C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in wcet analysis. In *Program Analysis and Compilation*, pages 12–52, 2006.

24 S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

25 J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

26 N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

27 N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *PLILP*, pages 207–218, 1991.

28 C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loops iterations. *RTS*, 18(2-3), May 2000.

29 C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), August 2002.

30 N. Holsti. Computing time as a program variable: a way around infeasible paths. In *WCET*, 2008.

31 N. Holsti, T. Långbacka, and A. Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *DASIA*, 2000.

32 L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of esterel programs. In *DAC*, pages 870–873, 2009.

33 L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of esterel specifications. In *CODES-ISSS*, 2008.

34 R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Journal on Software and System Modeling*, 10(3), 2011.

35 R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *ECRTS*, 2002.

36 R. Kirner, P. Puschner, and A. Prantl. Transforming flow information during code optimization for timing analysis. *Journal on Real-Time Systems*, 45(1-2), 2010.

37 R. Kirner and P. P. Puschner. Classification of wcet analysis techniques. In *ISORC*, pages 190–199, 2005.

38 J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *PSI*, pages 227–242, 2012.

39 Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

40 P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, 2009.

41 F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In *CC*, 1998.

42 A. Metzner. Why model checking can improve wcet analysis. In *CAV*, pages 334–347, 2004.

43 H. S. Negi, A. Roychoudhury, and T. Mitra. Simplifying wcet analysis by code transformations. In *WCET*, 2004.

44 C. Y. Park and A. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, 24:48–57, 1991.

45 P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst. (RTS)*, 1(2):159–176, September 1989.

46 P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1), 1997.

47 D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static wcet analysis of real-time task-oriented code in vehicle control systems. In *ISoLA*, pages 212–219, 2006.

48 F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES*, 2001.

49 I. Stein and F. Martin. Analysis of path exclusion at the machine code level. In *WCET*, 2007.

50 V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, pages 358–363, 2006.

51 L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for Matlab Simulink/Stateflow. In *ACES-MB*, 2009.

52 Esterel Technologies. *Scade Language Reference Manual*, 2011.

53 H. Theiling. Ilp-based interprocedural path analysis. In *EMSOFT*, pages 349–363, 2002.

54 S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, pages 625–632, 2003.

55 A. Vrchoticky. Compilation support for fine-grained execution time analysis. In *LCTES*, 1994.

56 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.

Multi-architecture Value Analysis for Machine Code*

Hugues Cassé, Florian Birée, and Pascal Sainrat

`surname@irit.fr`
Université de Toulouse
Institut de Recherche en Informatique de Toulouse (IRIT)
118 Route de Narbonne, F–31062 TOULOUSE CEDEX 9, France

Abstract

Safety verification of critical real-time embedded systems requires Worst Case Execution Time information (WCET). Among the existing approaches to estimate the WCET, static analysis at the machine code level has proven to get safe results. A lot of different architectures are used in real-time systems but no generic solution provides the ability to perform static analysis of values handled by machine instructions. Nonetheless, results of such analyses are worth to improve the precision of other analyzes like data cache, indirect branches, etc.

This paper proposes a semantic language aimed at expressing semantics of machine instructions whatever the underlying instruction set is. This ensures abstraction and portability of the value analysis or any analysis based on the semantic expression of the instructions.

As a proof of concept, we adapted and refined an existing analysis representing values as Circular-Linear Progression (CLP), that is, as a sparse integer interval effective to model pointers. In addition, we show how our semantic instructions allow to build back conditions of loop in order to refine the CLP values and improve the precision of the analysis.

Both contributions have been implemented in our framework, OTAWA, and experimented on the Malärdalen benchmark to demonstrate the effectiveness of the approach.

1998 ACM Subject Classification F.3.2 Program analysis

Keywords and phrases machine code, static analysis, value analysis, semantics

Digital Object Identifier 10.4230/OASIcs.WCET.2013.42

1 Introduction

Safety of critical embedded real-time applications needs to be verified in order to avoid catastrophic issues. This verification concerns not only functional features but also non-functional ones like temporal properties. The Worst Case Execution Time (WCET) is an important element of time properties. Its computation by static analysis is required at the machine level to get confident results.

Some tools, like OTAWA [3], provide a generic framework to perform these kinds of analyses whatever the underlying architecture is, including programming and execution models. In this article, we show how to adapt static analysis of data flow analysis to the machine code. More precisely, we have adapted and improved the circular-linear representation of integer values of [9][8] to any machine code using an architecture-independent language while maintaining fast convergence for the fixpoint computation.

* The research leading to these results has received funding from ANR under grant ANR-12-INSE-0001.

© Hugues Cassé, Florian Birée, and Pascal Sainrat;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 42–52

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Such an analysis is utterly important as its results can be used in the implementation of a lot of other analyses concerning control or data flow. For example, it can be used to compute the targets of indirect branches (pointer call or optimized `switch` implementations using indirection tables) or to identify statically code that is dynamically dead (typically in library functions where some conditions are always false because of the call context). Data flow uses include the analysis of data caches, of array accesses (bounding of the array range in memory), of the stack size (very important in embedded applications with small memory sizes) and infeasible paths by providing information on the program conditions.

This article is divided in 5 sections. In the first one, we present the abstraction of the machine language and its application to Abstract Interpretation (AI). The second section presents the CLP representation of values and a proposed refinement based on the decoding of branch conditions. Then, Section 3 gives the results of the experimentations and we provide in Section 4 a comparison with existing value analyses. The last section concludes the paper.

2 Value Analysis at Machine Code Level

In this section, we first present the abstraction of the machine instructions and how to use it to build an AI.

2.1 Independent Machine Language

As critical embedded real-time systems are running on very different architectures, OTAWA provides an abstraction of the machine instruction semantics. This avoids (a) to tie the analysis to a particular instruction set and (b) to focus only on the more useful part of the instruction behaviour to analyze the computation of integer values and addresses.

Table 1 Semantic Instructions.

Instruction (I)	Semantics ($update_I$)
`add` d, a, b	$d \leftarrow a + b$
`sub` d, a, b	$d \leftarrow a - b$
`shl` d, a, b	$d \leftarrow a \ll b$
`shr` d, a, b	$d \leftarrow a \gg b$
`asr` d, a, b	$d \leftarrow a \gg_+ b$
`set` d, a	$d \leftarrow a$
`seti` d, i	$d \leftarrow i$
`scratch` d	$d \leftarrow \top$
`cmp` d, a, b	$d \leftarrow a \sim b$
`store` d, a, t	$M_t[a] \leftarrow d$
`load` d, a, t	$d \leftarrow M_t[a]$
`if` c, a, i	$if\ a \neq c\ then\ ic \leftarrow ic + i$
`branch` d	$pc \leftarrow d$
`cont`	stop interpretation

$d, a, b \in Registers \cup Temporaries$
$i, s \in \mathbb{Z}$
$\ll$, $\gg$: logical shift left, right
$\gg_+$: arithmetical shift right,
$\sim$: comparison,
$c \in \{=, \neq, <, \leq, >, \geq, <_+, \leq_+, >_+, \geq_+\}$
that is, signed and unsigned comparators
pc: processor program counter,
ic: counter of semantic instrucions,
$M_s[a]$: memory cell of address a of type t
$t \in \{\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}\}$
where k in $\mathbb{Z}_k$, $\mathbb{N}_k$ is the number of bits
$\top$: undefined value

Unlikely to an RTL (Register Transfer Language) language, it is designed to make the static analyses easier and faster. Especially, it avoids actions internal to the microprocessor that are neither relevant, nor supported by the value analysis. This language, presented in Table 1, is composed of very simple a-la RISC instructions working either on machine

registers, or on temporaries. As some instructions may be very complex, one machine instruction may match a block of semantic instructions.

The semantic instructions set have been designed to be minimal and canonical and, therefore, to make their support in analysis easier: there is only one instruction performing a semantic function. Consequently, there are only addition, subtraction, shift and move operations. As our goal is only to support arithmetics on integer and particularly on address computation, we have included neither multiplication, nor division. These operations, often on powers of 2, are implemented more efficiently as shifts. In the same way, we have no equivalent of bit-to-bit operations like NOT, AND or OR as they do not produce precise results on interval analysis. The special instruction `scratch(r)` is needed to cope with these limitations: it means that the variable r is modified in a way that cannot be described by the semantic language. Indeed, there are always machine operations too atypical to be supported but a sound static analysis requires to mark r as modified even if the exact value cannot be expressed.

As a machine instruction is usually translated into a semantic instruction block, temporary variables are needed to pass computation intermediate results all along the block. They may take place of machine register in the semantic instructions but their life is bounded to the machine instruction block.

Another important concept to support is the conditional execution of some semantic instructions. In a machine instruction set, this is performed by using a comparison instruction followed by a branch instruction to modify the control flow accordingly. This scheme is currently supported by the semantic instructions but with some limitations.

First, there is a comparison instruction `cmp` that compares two values and stores the result in a target register (usually the status register of the underlying architecture). Then the comparison result is used by an `if` instruction. If the comparison from register a is equal to the c condition argument, the semantic instruction execution continues. Otherwise, the i following instructions are skipped and the execution continues just after them. An important outcome of such a structure is that the instruction block can exhibit several execution paths but no loop. This is an important property because the fixpoint computation induced by loops needs expensive computation time: several analyses are performed on the loop body.

`cont` is the second and last instruction handling the semantics control flow: it stops the execution of the current path. It must be noticed that a `cont` is implicitely assumed at the end of a block. The `if` and `cont` handle the control flow of the semantic instructions but there are also an instruction to represent control at the machine instruction level. The `branch` instruction informs that the machine control will change according to the address found in its arguments An important point to keep in mind is that this instruction does not modify the execution flow of the semantic instructions in the block: this one continues until the end of the block. It just denotes control flow changes in the machine instructions, i.e. modification of the PC.

Finally, `load` and `store` allow to load from, or store to, the memory. The first argument is the handled value while the second contains the address. The last one is the size of handled data in bytes.

This language makes easy and straight-forward the translation of most machine instructions but it may require more work when processing instructions as complex as multiple load-store to memory. An intermediate special computation phase is required but benefits from the instruction arguments, constant in the instruction code at the generation time. For example, a particular multiple-load instruction in memory gives the precise list of loaded registers and we can generate as many `load` semantic instructions as required. This genera-

Algorithm 1 `lmw` r_d, $k(r_a)$

$b \leftarrow [seti(t_1, k); add(t_1, t_a, t_1); seti(t2, 4)]$
for $i \leftarrow d$ *to* 31 **do**
  $b \leftarrow b :: [load(r_i, t_1, 4); add(t_1, t_1, t_2)]$
end for

```
seti(t1, 0)
add(t1, r1, t1)
seti(t2, 4)
load(r29, t1, 4); add(t1, t1, t2)
load(r30, t1, 4); add(t1, t1, t2)
load(r31, t1, 4); add(t1, t1, t2)
```

Listing 1 lmw r29, 0(r1).

tion is performed only once per instruction and makes the static analysis faster: semantic instruction blocks are simpler than the full translation inducing loops. This is illustrated by Algorithm 1 that shows the generation of semantics instructions list (between '[' and ']') for the PowerPC `lmw` instructions: starting from address $r_a + k$, it loads registers from r_d to r_{31}. As d, a and k are constant at the translation time, a particular instantiation, as shown at the right, just gives a sequence of loads.

2.2 AI with Semantic Instructions

The semantic instructions have been designed to promote static analyses and more particularly AI. AI [5] analyzes a program by abstracting the state S along the different execution paths. With machine code, AI is often performed on the Control Flow Graph (CFG), $G = V \times E$, where the vertices V represent Basic Blocks (BB), a block of consecutive instructions executed together, and edges, $E = V \times V$, the control flow between BB.

Algorithm 2 CFG Interpretation

$wl \leftarrow \{v_0\}$
while $wl \neq []$ **do**
  $v_i, wl \leftarrow wl$
  $s \leftarrow update(v_i, join(\{s_j / (v_j, v_i) \in E\}))$
  if $s \neq s_i$ **then**
    $s_i \leftarrow s$
    $wl = wl \cup \{v_j / (v_i, v_j) \in E\}$
  end if
end while

Algorithm 3 $update([I_0, I_1, ..., I_{n-1}], s_0)$

$s_r \leftarrow \bot$; $wl \leftarrow \{(0, s_0)\}$
while $wl \neq []$ **do**
  $(i, s), wl \leftarrow wl$
  if $i \geq n$ **then**
    $s_r \leftarrow s_r \cup s$
  else if $I_i = [\![if(c, a, d)]\!]$ **then**
    $wl \leftarrow wl \cup \{(i+1, s), (i+d, s)\}$
  else
    $wl \leftarrow wl \cup \{(i+1, update_I(I_i))\}$
  end if
end while

Algorithm 2 is a common implementation of the AI on the CFG. $v_0 \in V$ is the entry vertex of the CFG and the $s_i \in S^\#$ are the abstractions of the real state at the different program points. $S^\#$ is often a lattice with a smallest element, $\bot$, and greatest element, $\top$. $\bot$ is the initial value of s_i except for v_0 whose initial value is $\top$, that is, the more inaccurate value taking into account any possible state before the program execution. Algorithm 2 just ensures that the computation converges to a fixpoint, that is, the maximum of all possible values. This property is ensured by the existence of the lattice and the monotonicity property of any function handling the state.

This computation requires two analysis-specific functions. $update(v, s)$ emulates the effect of a basic block v on a state s while $join(s_1, s_2, ...)$ allows to combine different states coming from different paths. The semantic instructions are only used in the $update(v, s_0)$

function. For each machine instruction, the block of semantic instructions B is interpreted according to Algorithm 3. The different execution paths are supported with a working list wl containing pairs composed of the index of the current instruction and the current state. $update_I$ implements the computation effect on a state as shown in Table 1. Different execution paths are created when a `if` instruction is found: two pairs are pushed in wl for each possibility. When all execution paths have been computed, the resulting states are joined in s_r. As one may observe from Algorithm 3, the semantic instruction analysis is simple and straight-forward and re-uses directly the operators defined by the AI.

3 Proof of Concept: CLP analysis

For experimentation purpose, the semantic language has been used to implement a Circular-Linear Progression (CLP) analysis [9]. To push our semantics model even further, an analysis of conditions has been 0developed and applied to the CLP to tighten the precision.

3.1 Circular-Linear Progression Analysis

The CLP is an abstraction of integer values represented by a tuple (l, δ, m) denoting the set $\{n \in \mathbb{Z}\ s.t.\ n = l + \delta i\ \wedge\ 0 \leq i \leq m\}$, where $l \in \mathbb{Z}(n)$, $(\delta, m) \in \mathbb{N}(n)^2$. l is the base of the set, δ the increment, and m the count of increments on l to get the last point $l +_n \delta \times m$. The addition is performed on n bits (modulo 2^n), inducing a circularity on CLP and mimics the integer behaviour on the real hardware. The abstraction of a value k is easily obtained by the singleton $(k, 0, 0)$. while the top element $\top$ (a CLP that contains all possible values) is $(l, 1, 2^n - 1)$.

Performing CLP analysis on the machine code requires to abstract the semantics instructions on the CLP domain. For sake of brievity, only the `add` is given below but details on other operations can be found in [9]:

- $\{l_1\} + \{l_2\} = (l_1 +_n l_2, 0, 0)$
- $(l_1, \delta_1, m_1) + (l_2, \delta_2, m_2) = (l_1 +_n l_2, g, m_1 \frac{\delta_1}{g} + m_2 \frac{\delta_2}{g})$ where $g = \gcd(\delta_1, \delta_2)$.

The CLP allows to define the abstract states of the machine as a map from registers R and memory addresses A to CLP values, that is, $S : (R \cup A) \to clp$. AI operators are now defined by $U_S : V \times S \to S$ (update) and $J_S : S \times S \to S$ (join). J_S is naturally derived from the join function on the CLP , applied on the values assigned to registers and addresses.

The update function, U_S, is implemented as presented in the previous section. The abstract state S is applied to each machine instruction and, therefore, to each execution path of the semantics instructions. CLP values of the registers and of the memory are read or written by getting or setting them in the machine abstract state S.

3.2 Widening Function

To converge faster to a fixpoint, a widening function, $\nabla : S \times S \to S$, is useful. The example in Listing 2, a simple loop computing the sum of the elements of an array, allows to illustrate this. Listing 3 shows the translation of machine code of the loop header into semantic instructions.

At the first iteration of the loop, $p_0 = @t$, $@t$ being the actual address of array `t` in memory. At the second iteration of the loop, $p_1 = p_0 + 4 = @t + 4$. The widening ∇ is applied to these two CLP: $(@t, 0, 0)\nabla(@t + 4, 0, 0) = (@t, 4, 2^n/4 - 1)$. The resulting CLP is sound (it contains all possible values) and fast to obtain, but not very precise: next paragraph help to fix this.

```
int t[10];
int *p,int *q;
int s = 0;
q = t;
p = q;
while(p - 10 <= q)
{
        s += *p;
        p++;
}
```

Listing 2 Example of a simple C loop.

```
; r0 = 0x7fc4 (variable q)
seti t2,0x10
add t1,r31,t2    ; t1 = 0x7fc0
load r9,t1,0x4   ; r9 is p
seti t1,-0x28
add r9,r9,t1     ; r9 <- r9 - 40
cmp r71,r9,r0
if gt,r71,0x1
cont
seti t1,0x9c
branch t1
```

Listing 3 Translated Loop header.

3.3 Condition Filtering

In the example of Listing 2, we get the state $p = (@t, 4, 2^n/4 - 1)$, and a condition equivalent to $p - 40 > q$ ($10 \times sizeof(int)$ for a 32-bit machine). The edge remaining in the loop, taken if the condition is false, should provide as input state $p = (@t, 4, 2^{n-2} - 1)$ filtered by $p - 40 \leq q$ (inversed condition), i.e. $p = (@t, 4, 10)$. This would give an accurate state for the values inside the loop if we are able to build such a filter.

Unlike a condition written in a high-level programming language, the machine code does not provide a well-identified single expression for the loop condition. Instead, we must rebuild the link between variables (either registers or values in the memory) involved in the condition. In addition, one may remark that the same variable can be stored in many places, as registers or memory locations at the same time during the execution. Therefore, building the condition only on the register used in the comparison would make our approach very ineffective because it would ignore the aliasing existing between registers and memory. This is illustrated in Listing 3: the variable p is stored at the memory address `0x7fc0`, then loaded in $r9$. $r9$ is used again to carry out the result of $p - 40$. So the condition analysis must return two filters: `@0x7fc0` $> q + 40$ and $r9 > q$.

The algorithm proposed here tries to cope with both issues: the backward traversal of the semantics execution paths allows to collect the conditions of the `if` semantic instruction and to build back the aliases existing between registers and memories. Applied to the branch instruction of the loop header, the execution paths are sorted in two categories: the branching paths ending with a `branch` instruction and the continuing paths. The execution paths are then extended with other instructions of the loop header to completely form the computed condition. Each category applies a filter to the output state s of the loop header edge it matches (taken for the branch set, untaken for continuing set). As a category may contain several paths, the filters are applied on s separately and the result is joined by J_S.

Taking a path p (either branching or not), the filters are built by recording the condition induced by the `if` instructions and by rewritting the conditions when a computation instruction is found. To represent a condition, we use a simple parenthesed tree-based language where nodes are either constants c, registers r, memory places $@a$ or binary operators $\omega(e_1, e_2)$ where $\omega \in \{add, sub, ..., eq, ne, lt, ...\}$. As shown in Algorithm 4, the rewritting is performed backward (denoted by p^{-1}) from an empty set of conditions. `set` and `load` instructions are considered as creating an alias with their destination register and the filter is duplicated to generate condition for both places. The expression $f[x/y]$ means that the register x is replaced all over the filter f by y. Notice that, if a register computation is not involved in the condition building, the replacement have no effect on the filter f.

Algorithm 4 Building symbolic expressions

$f \leftarrow \{\}$
for all $s_j \in p^{-1}$ **do**
  $f \leftarrow filter[s_j]f$
end for

With *filter* defined by:

$$\begin{aligned} filter[\mathtt{if}(r, c, _)]f &= f \ \wedge \ c(r, \overline{r}) \\ filter[\mathtt{cmp}(c, a, b)]f &= f[a/r][b/\overline{r}] \\ filter[\mathtt{set}(c, a)]f &= f \ \wedge \ f[a/c] \\ filter[\mathtt{load}(c, a, _)]f &= f \ \wedge \ f[@a/c] \\ filter[\omega(c, a, b)]f &= f[\omega(a, b)/c] \end{aligned} \quad (1)$$

The application of the obtained filter f to an abstract state S is quite forward on a state s. For each CLP value stored in s whose reference, register or memory, is i, each reference $i' \in f$, $i' \neq i$, is replaced by its values in s, $f[s[i']/i']$ that allows, after simplification, to get a condition of the form $\omega(i, c)$ where $\omega \in \{ne, eq, lt, le, gt, ge\}$ and $c \in$ CLP. According to the actual operator ω, a CLP c_ω is obtained and intersected with the value of i: $s[i] \cap c_\omega$. For example, in Listing 3, the continuing path gives the filter $le(sub(r_9, 0x28), r0)$. Replacing the values of r_{31} and $r0$ by the matching CLP in s and simplifying gets $le(r_9, 0x7FC4 + 0x28$ and refines the value of r_9 by $s[r_9] \cap (-2^{31}, 1, 2^{31} + 0x7FC4 + 0x28) = (0x7FC4, 4, 10)$.

The application of the CLP to our semantic instructions has shown that (1) it is feasible to support such type of static analysis, and (2) there are different ways to use the semantic instructions as in condition filtering. We can hope that the semantic instruction is a valuable abstraction of the machine code instructions. In the next section, we try to evaluate the performances of this representation.

4 Experimentation

The value analysis presented in this paper has been implemented in the OTAWA framework [3] using its own internal AI engine. The evaluation has been performed with a 3-GHz, 2GB memory Linux machine on the classic Mälardalen benchmark [1] that contains a collection of programs covering different embedded real-time domains .

4.1 Analysis Precision

This first evaluation criterium concerns the precision of the performed analysis. We are not able to estimate the actual precision of the obtained measurements in terms of difference between the real values and the analyzed ones: (a) we have no other analysis that can be taken as a reference and (b) it is impossible to have precise values as soon as a program is using external inputs. The more visible trace of imprecision is the apparition of $\top$ values in the computation. Yet, it is hard to qualify the actual source of this imprecision as being naturally produced by the AI or an outcome of the intrinsic inefficiency of our analysis. In turn, the last issue may be decomposed in two causes: the lack of expressivity of semantics instructions or the limits of the abstraction of the CLP analysis.

Whatever, the last three columns of Figure 2 show the number of non-$\top$ values obtained for different items: each column represents the ratio of non-$\top$ values on the total of values of the measured items, the bigger is the better. The *values* column displays the number of set values generated by `set` and `store` semantic instructions with an average of 42.30% of non-$\top$ values. In the absolute, the result is a bit disappointing but (1) to our knowledge, no such statistics have been published to compare with and (2) the large variation between benches (from 2.47% to 99.98%) shows that the effectiveness has a big dependency on the type of

Table 2 Analysis Execution Time.

Program	Time (μs)	Dynamic mach (i/s)	Dynamic sem (i/s)	Static mach (i/s)	Static sem (i/s)	Value Precision values (%)	Value Precision addrs (%)	Value Precision filters (%)
adpcm	31530	219600	495908	39861	204598	2.47	15.39	72.34
bs	1190	103361	228571	97560	141176	53.85	89.09	33.33
bsort100	2680	117537	269029	63492	110074	20.75	17.31	33.33
cnt	2590	214285	462548	50450	188030	36.17	46.51	100.00
compress	25610	85591	196134	48813	88481	8.68	27.42	49.30
cover	50040	59492	159852	156533	81434	8.66	52.94	100.00
crc	5160	230232	446511	47979	207751	10.04	79.25	86.52
duff	1100	97272	233636	271028	441818	38.46	58.70	60.87
expint	3290	115805	272644	70866	132218	47.37	71.43	80.00
fac	30	1400000	3233333	1000000	4533333	77.78	95.00	100.00
fdct	1330	958646	2051127	8627	1057142	20.98	99.42	100.00
fft1	42220	247418	485243	17231	87162	24.77	61.90	16.03
fibcall	910	98901	252747	100000	142857	74.19	100.00	100.00
fir	3330	104504	268168	45977	97897	34.48	90.50	16.00
insertsort	1070	241121	481308	34883	202803	47.37	72.22	33.33
janne_c	2360	72457	188559	93567	74576	59.52	87.21	77.78
jfdctint	1940	718556	1690721	10043	598453	6.72	91.57	100.00
lcdnum	4040	73267	177970	162162	92574	30.99	50.45	100.00
lms	14590	202604	385263	33491	125222	49.47	91.25	81.94
ludcmp	12660	138151	280489	29731	77725	33.16	82.85	29.65
matmult	4140	238647	503864	38461	145893	23.33	29.44	100.00
minver	16320	145220	306250	32911	87438	13.49	67.93	29.00
ndes	27550	167622	369546	33347	111724	52.28	74.46	74.39
nsichneu	205180	98450	209182	37425	104664	11.98	89.68	29.80
ns	8060	99503	206203	24937	29528	83.58	89.40	61.29
prime	2500	162800	362000	100737	237200	70.75	100.00	50.00
qsort-exam	5860	139761	289419	41514	150170	13.46	63.50	16.85
qurt	6320	154113	305537	61601	233386	44.81	100.00	100.00
recursion	30	800000	2033333	1714285	4066666	69.23	100.00	100.00
select	5150	136893	283300	51063	149708	19.75	66.42	36.17
sqrt	1610	91304	182608	74829	88198	50.00	100.00	100.00
statemate	48770	157227	307709	40688	109883	3.81	90.85	89.02
st	7680	213411	430989	58572	245833	10.12	19.33	100.00
ud	7630	144429	300262	37205	114285	33.59	81.25	26.72
average		242593	539705	139114	428232	34.88	72.14	67.17

the program. Some benchmarks like *crc* or *adpcm* give particularly bad results because they are performing a lot of operations unsupported by our semantics instructions, respectively, bit-to-bit and floating-point operations.

However, our analysis works better with address computation (74% on a mean, column *addrs*) and condition filtering (68% on a mean, column *filters*). The *Addresses* column evaluates the non-$\top$ addresses used in `load` and `store` instructions while the *filters* column

evaluates the number of non-$\top$ filtered conditions. The obtained results are not perfect but they meet the needs of accuracy required by subsequent analyses like data cache, control flow analysis, infeasible path, etc. As for *set values*, the worst results are obtained on benchmarks using unsupported operations of the semantic language.

4.2 Computation Time

The six first columns of Table 2 show runtime performances of the analysis. We call *static* estimation the performance evaluation based on the count of instructions found as-is in the program once loaded in memory. On the opposite, *dynamic* instructions are counted during the AI. Both estimations are different because a single instruction may be interpreted several times before reaching a fixpoint: this is particularly the case of instructions contained in loops. The left column displays the experimented program, the next one the analysis time in μs (user time mean after 1000 iterations). The following two columns show the rate in machine and in semantic instructions, for dynamic estimation and the last two columns are the same for static.

The static estimation gives an insight of the capacity of the analysis to face to a raw program. For example, the average of about 139000 instructions / s states that we can process quickly (in less than 1 second) a program of nearly 512KB of code for a PowerPC architecture or any pure 32-bit RISC architecture.

On the other hand, the dynamic performances give a more straight-forward estimation of the real rate of the analysis, 242,594 machine instructions/s and 539,705 semantic instructions/s, taking into account the structure of the program. Hopefully, both measures are quite good although there is a lot of variability according to the benchmarks. Such performances let place for improving analysis without being blocked by intractable computation time.

5 Comparison with Existing Work

A lot of work on value analysis has been done for different purposes. The foundation of abstract AI was motivated by the interval analysis $[l, u]$ of each variable on Pascal-like languages [5]. In addition to the domain, the performed analysis is also different because of the widening operation. To speed up the convergence to the fixpoint, widening and narrowing operators are applied, that requires several passes of analysis and increases the computation time. To our knowledge, although there is very few documentation on this [4], the tool ait [2] is also using such an analysis to exhibit register values at program points but is applied to machine language.

Yet, the interval of values $[l, u]$ does not fit well with address representation. They include too many false values while the addresses are usually aligned to a multiple of data sizes and cause the union of too many false data and reduce the precision. Ermedhal et al in [6] improve this using a modulo analysis to identify more precisely values inside an interval. Yet, their domain is more complex than ours, $([l, u], (i, c))$ that identifies the set of values matching $[l, u] \cap \{i + n \times c, n \in \mathbb{Z}\}$. Moreover, this domain makes the analysis more costly in computation time.

S. Rathijit [8][9] simplifies a lot this representation using only three integers, (l, u, δ). i, the argument of Ermedhal's representation is easily replaced by finding the lowest value of the set to get a more precise l. Then, we have improved Rathijit's work to have canonical representation of these values. Indeed several representations for a same set exist because $u - l$ is not required to be a multiple of δ. Our triplet (a, δ, n) ensures the uniqueness of the representation as it forces $u = a + n \times \delta$. This property makes easier some operations of

the value arithmetics like equality tests. In addition, unlike Rathijit that seems to support only ARM, our machine code abstraction, avoids to tie our analysis to a specific machine instruction set and we are using the conditions to accelerate convergence of fixpoint of the analysis and improve the precision in presence of selections.

In the domain of semantics languages, ALF [7] is supported by several compilation and WCET tools. It allows to represent completely a program while our language is just an overlay on the CFG. More comprehensive and more expressive than our semantics instructions, it seems to be more memory- and time-consuming for analyses. Architecture Description Languages like SimNML, Lisa, etc are also good candidate to express semantics of machine code. But we think that their granularity would make the analyses too costly and that some concepts are too hard to extract: for example, in a microprocessor, the comparison result is represented as a set of bits obtained by a combination of operand bits whose usage in analysis is not straight-forward.

6 Conclusion

The contribution of our paper is twofold. First, we propose a language to abstract usual machine code semantic and show how it may be involved in abstract AI. The overall outcome is that the analyses based on this language becomes portable on any microprocessor supported by our semantic language. For example, in the OTAWA framework, it has been already successfully used to describe several RISC instruction sets like PowerPC. We plan to quickly apply the semantics instructions to Sparc, TriCore and possibly to the x86 instruction set.

Second, we have improved existing value analysis, based on CLP domain, for speed of computation. The changes include a reformulation of the domain leading to canonical values and therefore simplification of operation abstraction. The second improvement concerns the use of conditions to speed up the convergence of fixpoint computation.

The resulting analysis rate is quite fast, which will be valuable in the future in order to fix medium precision results. We think that the precision problem is not inherent to the value analysis algorithm but, instead, to the limitation of the semantic language. So, we plan to extend it with bit-to-bit operations, integer multiplication and division as well as floating-point operations. The latter extension cannot be done in the frame of CLP because the notion of modulo does not fit well with float value: we have to fall back to usual interval analysis. This means a state abstraction with heterogenous content.

Finally, we would like to exploit the results of this value analysis to extend the OTAWA framework. The semantic language has already been used in OTAWA to implement data cache analysis and stack analysis but we can exploit the value analysis results to improve control flow analyses like detection of infeasible paths, dynamically dead code, switch decoding, indirect pointer call analyzes. As the latter analysis does not fit well with CLP, we have also to introduce set of addresses in our domain and it remains to identify which values should be CLP and which ones should be sets. In a more generic way, we have to replace our naive implementation of state to improve our value representation to support heterogeneous data, to maintain fast computation and to waste as less memory as possible.

References

1 Mälardalen benchmarks. `http://www.mrtc.mdh.se/~projects/~wcet/~benchmarks.html`.

2 ait tool, 2005. `http://www.absint.com/ait/`.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

4 C. Ferdinand, R. Heckmann, and D. Kästner. Static Memory and Timing Analysis of Embedded Systems Code. In *Proceedings of the IET Conference on Embedded Systems at Embedded Systems Show*, 2006.

5 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1977.

6 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation and Invariant Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis*, 2007.

7 J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF – A Language for WCET Flow Analysis. *WCET'09*, 30 June 2009.

8 S. Rathijit and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. 2007.

9 S. Rathijit and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, 2007.

The Auspicious Couple: Symbolic Execution and WCET Analysis*

Armin Biere[1], Jens Knoop[2], Laura Kovács[3], and Jakob Zwirchmayr[2]

1 **Johannes Kepler University Linz, Austria** `biere@jku.at`
2 **Vienna University of Technology, Austria**
`[knoop|jakob]@complang.tuwien.ac.at`
3 **Chalmers University of Technology, Sweden** `laura.kovacs@chalmers.se`

Abstract

We have recently shown that symbolic execution together with the implicit path enumeration technique can successfully be applied in the Worst-Case Execution Time (WCET) analysis of programs. Symbolic execution offers a precise framework for program analysis and tracks complex program properties by analyzing single program paths in isolation. This path-wise program exploration of symbolic execution is, however, computationally expensive, which often prevents full symbolic analysis of larger applications: the number of paths in a program increases exponentially with the number of conditionals, a situation denoted as the path explosion problem. Therefore, for applying symbolic execution in the timing analysis of programs, we propose to use WCET analysis as a guidance for symbolic execution in order to avoid full symbolic coverage of the program. By focusing only on paths or program fragments that are relevant for WCET analysis, we keep the computational costs of symbolic execution low. Our WCET analysis also profits from the precise results derived via symbolic execution.
In this article we describe how use-cases of symbolic execution are materialized in the r-TuBound toolchain and present new applications of WCET-guided symbolic execution for WCET analysis. The new applications of selective symbolic execution are based on reducing the effort of symbolic analysis by focusing only on relevant program fragments. By using partial symbolic program coverage obtained by selective symbolic execution, we improve the WCET analysis and keep the effort for symbolic execution low.

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

Keywords and phrases WCET analysis, Symbolic execution, WCET refinement, Flow Facts

Digital Object Identifier 10.4230/OASIcs.WCET.2013.53

1 Introduction

Symbolic execution can analyze a program with high precision, by using symbolic instead of concrete input values of the program. Programs are symbolically executed path-wise, and each program path is analyzed in isolation. This, however, comes at the price that every program path needs to be symbolically executed in order to infer results that are valid for the entire program. In other words, a *full symbolic coverage* of the program is needed for verifying program properties using symbolic execution. As the number of paths increases

* This research is supported by the FP7-ICT Project 288008 T-CREST, the FWF RiSE projects S11408-N23 and S11410-N23, the WWTF PROSEED grant ICT C-050, the FWF grant T425-N23, and the CeTAT project of the TU Vienna.

© Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr;
licensed under Creative Commons License CC-BY

exponentially with the number of conditionals in the program, computing full symbolic coverage for larger applications is in practice not realistic. Applications of symbolic execution therefore only explore relevant parts of the program behavior and compute a *partial symbolic coverage* of the program.

Such a compromise between precision and computability is also present in the Worst-Case Execution Time (WCET) analysis of programs. Namely, a successful WCET analysis requires a balance between the speed and the precision of the deployed analysis. Precision of the analysis is gained by applying powerful program analysis techniques that gather information about the program and pass it to further analysis- and computation-steps. Precision of the analysis yields tight WCET estimates, however, at the cost of high computational effort; this sometimes prevents the analysis to terminate within a given time-limit. Precision of the WCET analysis is therefore often traded for its speed: faster analysis with likely imprecise WCET estimates is preferred to a precise but slow one. Following this compromise, automated methods for refining imprecise WCET results into tighter ones are needed in the WCET analysis of programs.

In this article we argue that combining symbolic execution with traditional WCET analysis yields an efficient and precise method for computing WCET estimates. We show that, for using symbolic execution in WCET analysis, a partial symbolic coverage of the program is sufficient to tighten and, eventually, prove the computed WCET bound of the program to be precise. We do so by applying *selective symbolic execution* over program parts and avoid the path explosion problem of traditional symbolic execution. To this end, we use costly symbolic execution only for those parts of the program that influence the WCET estimate. Our WCET-guided symbolic execution is a precise selective symbolic execution for *relevant* parts of the program, and avoids the computational overhead of full symbolic execution. Our workhorse in this article is the r-TuBound toolchain [16].

We extend r-TuBound with symbolic execution (Section 3) and present three existing applications of symbolic execution in r-TuBound for WCET analysis (Section 4):

- We use symbolic execution in r-TuBound on *selected program fragments* to analyze programs which could not be analyzed by r-TuBound due to a too restrictive programming model of [15];
- We deploy symbolic execution in r-TuBound to *compute loop bounds*. This extension allows r-TuBound to calculate loop bounds in cases where it has previously failed;
- Based on the implicit path enumeration technique (IPET) [19], we use the result of an initial WCET analysis and apply symbolic execution in r-TuBound to *tighten initial WCET estimates and eventually prove these bounds precise*, by applying the work of [17].

Based on our current use of symbolic execution in r-TuBound, we also discuss further applications of symbolic execution for the WCET analysis of programs (Section 5). These new directions rely on partial symbolic coverage of the program and include:

- inferring *precise execution frequencies* for loops with conditionals;
- generating *WCET path test-cases* used in measurement-based WCET analysis tools;
- automated support for *mode-sensitive analysis* of programs after an initial (IPET-based) WCET analysis.

We believe that the WCET applications proposed and discussed in this article encourage the further use of symbolic execution in WCET analysis. The symbolic execution extensions that are already implemented in r-TuBound were successfully applied to examples coming from the WCET tool challenge [10]: WCET estimates were tightened and the cost of symbolic execution was low. We are confident that the overhead for the proposed applications of symbolic execution in WCET analysis can be kept low, while providing valuable information about the program.

2 Preliminaries

In this section we give an overview of the main ingredients of symbolic execution and WCET analysis. For more details, we refer to [3, 6] and [15, 18], respectively.

Symbolic Execution. Symbolic execution uses symbolic instead of concrete input data to symbolically execute a program. To do so, input variables of the program are assumed to be "symbolic," which means that they can have an arbitrary value (conforming to the specified data-type). If a conditional statement splits the control-flow of the program, symbolic execution follows both successor edges of the conditional, restricting possible values of symbolic variables according to the condition. For example, if a conditional executes the *true*-edge of the condition only if a variable has a certain constant value, then symbolic execution assumes the constant value for the variable when following this edge. Thus, symbolic variable values are restricted by path conditions or assumptions involving the respective variable. This allows to track complex constraints for each variable and use solvers, such as [5], to reason about the derived constraints.
Symbolic execution of programs with conditionals and loops often leads to the path explosion problem, as the number of paths needed to be symbolically executed increases exponentially with the number of conditionals in the program. Hence, full symbolic coverage of larger applications is infeasible in practice. The problem of path explosion can be addressed in different ways, e.g., by using heuristics for computing only partial symbolic coverage of the program, for instance in the context of test-case generation and bug-hunting.

WCET Analysis. A static WCET analysis toolchain typically includes several high-level analyses that gather so-called flow fact information about the program. Essential flow facts include loop bounds and execution frequencies of conditional edges in the program. When computing WCET estimates, the underlying hardware architecture needs to be analyzed for inferring execution times of program blocks. Additional hardware features, such as cache-configuration and pipeline layout, also need to be taken into account. Precision of WCET bounds denotes, in this article, that any over-estimation of the WCET is due to an imprecise hardware modelling and not due to infeasible paths.

With the block execution times computed for the program, various techniques can be applied to find the path that exhibits the WCET of the program. One of the most common approaches is the implicit path enumeration technique (IPET) [19]. It is applied to the control flow graph (CFG) of a program and relies on the fact that each program execution satisfies the following flow properties: (i) a program execution executes the entry point of the program once and (ii) other program blocks are executed as often as their predecessor blocks. Therefore, any program block following the entry point is executed once, unless it appears in a conditional or loop statement. For a conditional (iii) the total sum of the execution frequencies of its conditional blocks (denoted the `true`- and `false`-block of the conditional) coincides with the execution frequency of the predecessor block of the conditional, which is the condition-block. For blocks inside loops, (iv) the execution frequencies are multiplied by the loop bound. Hence, when applying IPET, loop bounds are assumed to be supplied as flow-facts.

To apply IPET, the program is represented as an integer linear program (ILP) where each program block is modeled by an ILP variable that has the block execution time associated with it. An ILP-solver [2] is then used to solve the flow problem (i)–(iv) specified above. By using an ILP encoding on execution frequencies, the solution of the corresponding ILP

problem assigns values to the ILP variables, that is, execution counts of program blocks. The WCET estimate for the program is then obtained by maximizing the sum of the products of execution frequencies and block execution times for each block.

As flow facts about programs are not always precise (e.g. loop bounds are not exact but are over-approximated), the ILP encoding of the program usually encodes numerous spurious program paths. Some of these spurious program paths might yield high execution times. Therefore, the WCET estimate computed from the ILP solution is usually an over-estimation of the actual WCET: its precision crucially depends on the quality of additional flow facts supplied to IPET. For example, supplying additional flow facts that specify valid execution frequencies for conditional blocks can result in a tighter WCET estimate.

3 Selective Symbolic Execution in r-TuBound

In this section we describe our extensions to the r-TuBound toolchain [16], by integrating symbolic execution into r-TuBound. The common theme of all these extensions relies on a selective use of symbolic execution for timing analysis, instead of symbolically executing the whole program.

r-TuBound applies high-level analyses on the source level and calculates WCET estimates using a low-level analyzer. In a nutshell, the main steps of r-TuBound are as follows. Given a program with loops written in a restricted class of C, r-TuBound deploys interval and points-to analysis to derive bounds, called loop bounds, on the number of loop iterations. The source code, annotated with the results of these analyses, is then compiled by a WCET aware compiler. The resulting assembly is analyzed by the WCET analyzer CalcWCET167 of the Infineon C167 microprocessor [14]. It applies the IPET approach, solves the resulting ILP problems and derives WCET estimates as outputs.

Relying on the infrastructure of [16], we extended r-TuBound by symbolic execution. Figure 1 shows the current workflow of r-TuBound, where the colored components correspond to the new extensions. We refined the loop bound computation step of r-TuBound by exhaustive symbolic execution for loops, implemented in the `r-Loopbounds` step of Figure 1. We also added a *selective symbolic engine* to r-TuBound for deriving tight WCET bounds, listed as the `Selective SE` step of Figure 1. In the rest of this section we describe the integration of symbolic execution in r-TuBound.

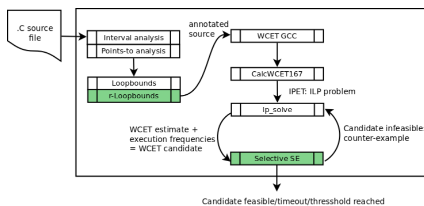


Figure 1 Architecture of the WCET toolchain. Colored parts rely on symbolic execution.

Symbolic Execution and r-TuBound. We use the symbolic execution engine of [3] in r-TuBound to construct a precise memory-model of a program. Given an input program (written in C), the program is first parsed and stored as an abstract syntax tree in the code-list. The code-list is then further processed: program paths are extracted and are symbolically executed by writing and reading the symbolic representation of the program memory. As a result, a representation of the symbolic program execution is obtained as a set of satisfiability modulo theory (SMT) formulas, where the SMT formulas are expressed in theory of bit-vectors and arrays. Verification conditions, expressing runtime and memory-access properties, are generated to guarantee runtime- and memory-safety of the program (e.g. does not dereference `NULL`). Properties that hold in the symbolic representation of

a program are also guaranteed to hold in the actual program. Instead of symbolically executing all paths in the program, selective symbolic execution allows to execute paths selectively by supplying a sequence of branching decisions that encode executions of the program. The branching decisions are extracted from the IPET solution. These decisions allow to iteratively select and symbolically execute WCET paths. Precise constraints are only inferred about the WCET path, reducing the costs for symbolic execution.

4 Precise WCET Analysis without Path Explosion in r-TuBound

Symbolic execution infers precise program properties that can further be used in an IPET-based WCET analysis. Nevertheless, symbolic execution comes with the cost of analyzing each program path, a practically infeasible task for large programs with loops and conditionals. To avoid this problem, when using symbolic execution for WCET analysis in r-TuBound, we identify relevant program parts, that is program fragments for which symbolic execution is necessary to be applied. More precisely, we apply symbolic computation in the following three scenarios: (i) analyzing reduced program fragments in isolation of the entire program, by reasoning about single statements in loop bodies, as well as about loops and nested loop structures, (ii) deriving loop bounds on the number of loop iterations, and (iii) analyzing a small number of paths for refining and proving precise WCET bounds. For doing so, (i) relies on the programming model of [15], (ii) makes uses of [3] and is restricted only to the programming model of the underlying symbolic execution engine, and (iii) is based on the theoretical framework presented in [17].

In what follows, we overview the relevant parts of symbolic execution in each of the above scenarios of r-TuBound, and illustrate our work on the example of Figure 2. Based on the current applications of symbolic execution in r-TuBound, in Section 5 we will outline new and ongoing applications of symbolic execution for WCET analysis.

```
1: int main (int flag) {
2: int i;
3: for(i = 0; i < 5; i++)
4:   if(i == 4 && flag) {
5:     i = 0;
6:     flag = 0;
7:  }
8: }
```

Figure 2 Our running example. All data is assumed uninitialized.

(i) Analyzing reduced program fragments. We use the symbolic execution framework of [3] to verify arithmetic properties about one or more conditional updates of the loop counter in a loop. Symbolic execution is appropriate in this setup as interval analysis often lacks sufficiently precise analysis results. By using symbolic execution, in the current version of r-TuBound we can verify arithmetic properties about loop counters without the need of deriving tight intervals for the values of loop counters. Even more, we are able to handle a more general programming model than the one used in [15]. Namely, we can analyze loops whose conditional updates are arbitrary expressions in the combined theory of linear arithmetic, bit-vectors and arrays, whereas [15] was restricted to the theory of linear arithmetic. If the such derived arithmetic properties are proved to be correct by using [3], the loop bound computation step of r-TuBound can safely be applied. Our use of symbolic execution also allows to merge conditional updates of the loop counter into a so-called *combined minimal update*, which then yields a tighter loop bound, and hence a tighter WCET estimate.

▶ **Example 1.** The loop analysis step of [15, 16] fails to compute a bound for the loop in Figure 2. This is so because the conditional update to the loop counter i (in line 5) violates the computed loop bound in cases when the loop counter is reset, i.e. when `flag` is true. In Figure 2, i increases in the loop header. Therefore, the `r-Loopbounds` step of r-TuBound

symbolically executes the conditional update for an arbitrary (i.e. symbolic) loop iteration and verifies that the conditional update can only increase the value of i. If this property is violated, it is not safe to compute a loop bound using the techniques of [15, 16] implemented in the `Loopbounds` step of Figure 1. In the example in Figure 2, the property is false, the conditional update can decrease the loop counter, and therefore no loop bound is computed.

(ii) Loop bound computation. If the loop bound computation step of [15] in r-TuBound fails, we apply exhaustive symbolic execution of the *reduced* program in r-TuBound. As a reduced program we consider the program that only contains the loop under study together with relevant variable declarations, i.e. the variables used in the loop. Variable values are treated as symbolic, with the exception of the loop counter. Additional information, such as intervals for variable values or program slices, can also be supplied in this step to improve the precision of the loop bounds. A supplied time-limit guarantees termination of the approach. The reduced program is symbolically executed in r-TuBound, where r-TuBound initially sets the loop bound to 0. If symbolic execution reports that the negation of the loop condition is unsatisfiable on the (unwound loop) path, the loop bound is increased by one. Upon termination within the time-limit, no execution of the program exhibits a higher loop bound. Such a use of symbolic execution in r-TuBound is especially useful when bit-precise reasoning is required.

▶ **Example 2.** The approach of [15] cannot derive a loop bound for Figure 2. Therefore, (bounded) exhaustive symbolic execution is applied to only analyze the program loop. By using symbolic execution in r-TuBound, we derive 9 as the exact loop bound of Figure 2.

(iii) Deriving precise WCET bounds. The WCET analysis approach presented in [17] relies on the tight combination of a symbolic execution engine and a WCET analyzer. It first applies an IPET-based WCET analysis that yields an ILP problem that encodes constraints on the program flow and a WCET estimate. Next, symbolic execution on single program paths is applied in order to infer constraints that allow tightening and ultimately proving the WCET bound precise. For doing so, the ILP solution describing the execution frequencies of program blocks and the ILP problem is analyzed, and one or more spurious program execution traces exhibiting the WCET are identified. These execution traces are then excluded from the set of possible program executions, by adding a new ILP constraint to the ILP problem. The resulting new ILP problem is then used in the next iteration of the approach, by again applying IPET in combination with symbolic execution. In each iteration either a new and lower WCET estimate is derived or a program trace exhibiting the old WCET is obtained. In the latter case, the computed WCET is the actual WCET of the program, and the algorithm terminates. The computed WCET is precise wrt the underlying hardware-model.

Note that in [17] only paths extracted from the ILP solution are symbolically executed, avoiding thus the full path explosion problem. problem of symbolic execution. The WCET estimates serve here as a measure for the *relevance* of paths: they allow to select relevant paths that need to be symbolically executed in order to tighten the WCET.

We implemented the approach of [17] in r-TuBound and describe it here. The program is parsed and the IPET approach is applied. An ILP problem is next obtained and solved, by using the ILP solver lp_solve of [2]. The obtained ILP solution encodes a WCET estimate of the program. Further, the execution trace specified by the ILP solution is (re)-constructed. We encode execution traces as sequences of branching decisions, where branching decisions are obtained as follows. For each conditional block, i.e. a program block

with jump-instructions to other blocks, the execution frequency of the jump-targets specifies which block is assumed to be executed on the path exhibiting the WCET estimate. If the condition evaluates to `false`, the `else`-block of the conditional is executed. Thus, the ILP solution specifies an execution frequency of 0 for the `then`-block and an execution frequency of 1 for the jump-target, the `else`-block. The inferred branching decision is `f`.

If both edges of a conditional have an execution frequency ≥ 0 in the ILP solution, both program blocks of the conditional are executed. In this case, the WCET candidate encodes multiple actual program executions. Hence, from the ILP solution, one or more program paths exhibiting the WCET estimate can be constructed. We therefore refer to program paths exhibiting the WCET as *WCET trace candidates.* If one of these is feasible, the computed WCET bound is proven precise and yields the actual WCET of the program in the underlying hardware model. If all of them are infeasible, the ILP problem of IPET can be refined and a tighter WCET estimate can be computed. WCET trace candidates in r-TuBound are expressed as SMT formulas in the combined theory of bit-vectors and arrays, and the SMT solver Boolector [5] is used to check their feasibility. In our current r-TuBound implementation we rely on a manually constructed mapping between the assembly analyzed with CalcWcet167 and the source of the application. In other words, we manually verify that the source and assembly exhibit a compatible branching behaviour. Construction of this mapping can be omitted when symbolic execution is performed on the binary level.

▶ **Example 3.** The initial ILP solution derived from the ILP problem of IPET (using the loop bound 9) specifies the execution of the conditional block in each iteration of the loop in Figure 2. The WCET trace candidate extracted from the ILP solution encodes exactly one program path; this path executes the conditional block 9 times. This WCET trace candidate is specified by the following sequence of branching decisions `t`...`t` (9 times `t`), where `t` denotes the true-edge of the conditional statement. By symbolically executing this WCET trace candidate, we derive the infeasibility of `t`...`t`. Thus, an additional ILP constraint is constructed to exclude this WCET trace candidate from the ILP problem. The new ILP problem is solved again, yielding a tighter WCET estimate and new WCET trace candidates. This process is iterated until a feasible WCET trace candidate is found. In Figure 2, a feasible WCET trace candidate is derived after 8 iterations. As a result, the exact execution frequency of the `true`-block of the conditional is inferred and constrained to 1. In a simplified scenario where execution of each program instruction takes 1 time unit (t), the actual WCET of the program is then derived to be $40t$. The WCET is derived by summing up the execution times (a)-(c): (a) the initialization in the loop header (`i=0`) takes $1t$; (b) Among the execution frequencies of loop iterations, based on the derived execution frequency of the conditional statement, the following case distinction is made: for 8 loop iterations, an iteration takes $4t$, $1t$ is the evaluation of the loop condition `i<5`, $2t$ are needed to execute the condition `i==4 && flag` of the conditional (two instructions) and $1t$ is taken for the loop counter increment `i++`. All together, these eight loop iterations take $32t$. One loop iteration, namely the one in which the conditional statement is executed, requires $6t$ to be executed. When compared to the previous cases, the additional $2t$ result from the execution of the true-block of the conditional. (c) The last evaluation of loop condition $i < 5$ after 9 loop iterations takes $1t$.

Summarizing, the applications (i)–(iii) discussed above share a common approach: instead of symbolically executing the entire program, selective symbolic execution is performed only on fragments or single paths of the program in order to prevent symbolic execution from running into the path explosion problem.

5 Further Applications of Symbolic Execution for WCET

In this section, we discuss three additional applications of symbolic execution in WCET analysis, by using the symbolic execution framework presented in Section 3. Similarly to Section 4, these applications apply only partial symbolic coverage of the program. The material presented in this section is work-in-progress and requires further experimentation.

Precise execution frequencies for loops . When the loop bound computation techniques of [15] fail, exhaustive symbolic execution of the loop is applied as described in Section 4(ii). The application scenario of Section 4(ii) can be further extended to compute execution frequencies for conditional blocks inside the loop: by applying exhaustive symbolic execution, a loop bound and feasible WCET traces are derived. The execution frequencies of program blocks in the feasible traces is also obtained. In case of multiple feasible traces, a set of execution frequencies is inferred for each block. We then set the execution frequency of a block to be less or equal to the highest and greater or equal to the lowest value among its set of possible execution frequencies, and use this value in the ILP encoding of the program. To ensure that the such chosen execution frequency is precise, we use the approach of Section 4(iii) and iteratively refine the execution frequency of each edge inside the loop.

▶ **Example 4.** Consider Figure 2 again. An initial IPET-based WCET analysis (without additional flow facts) sets the execution frequency of the `true`-block of the conditional to 9. By applying our approach, we use exhaustive symbolic execution on the loop of Figure 2. As a result, we derive the maximum execution frequency of 1 for the `true`-block of the conditional. Using this additional flow fact in the initial IPET-based WCET analysis, the precise WCET of the program is also derived.

(ii) WCET path test-cases. Our implementation of Section 4(iii) in r-TuBound can also be used to generate WCET path tests-cases. The approach of Section 4(iii) already extracts and symbolically executes WCET trace candidates. A symbolic execution engine can be used to generate concrete program inputs from the trace that is symbolically executed, forcing actual executions of the program to follow the same trace. The program inputs for feasible WCET trace candidates thus represent program inputs which lead to the execution of the actual WCET path, that is, a concrete program path exhibiting the WCET. The program inputs generated by symbolic execution can be used by hardware-aware dynamic WCET analyzers, see e.g. [20], to take additional, hardware-dependent, time measurements. We refer to such analyzers as measurement-based WCET analyzers. The generated test-cases can help measurement-based WCET analyzers to derive relevant timing behavior of the application on the WCET path. At the same time, the measurements can be used as feedback about the precision of static analyzers: little variation between the statically computed WCET estimate and the measurements on the WCET path are an indication of precision of the static analyzer. Even more, the statically calculated WCET estimate must never be below the WCET value reported by the measurement-based analyzer.

▶ **Example 5.** Consider Figure 2. The WCET analysis of Section 4(iii) infers the WCET path to execute the `true`-block of the conditional once, when `i` is 4 and when the value of the symbolic variable `flag` is assumed to be `true`. Based on the symbolic execution of this WCET trace, the symbolic execution engine in r-TuBound generates a test case which initially assumes `flag` to evaluate to `true`. Supplying this input to a measurement-based WCET analyzer that runs the actual program allows to take measurements on the WCET path.

(iii) Mode-sensitive analysis. The symbolic execution engine of r-TuBound used in Section 4(iii) can further be used to support automated *mode-sensitive* WCET analysis. Modes characterize a certain state that the program is executed in. For example, the program could be in a "normal operation," an "initializing" or an "error" mode. Given a program, its precise WCET is derived using the method of Section 4(iii). Assume now that the program is modified, e.g. , by setting a program flag resulting in a different mode of execution. The approach of Section 4(iii) will then automatically recompute the WCET for the modified program. This mode-sensitive behaviour can be observed in functions from the Mälardalen benchmark suite of [10], where the flags (`init`, `found`) control the execution mode and thus the WCET path. These control variables are not yet found automatically in r-TuBound. However, simple methods could be used to identify these flags, for example by checking whether two conditions are mutually exclusive. We leave the integration of r-TuBound with such techniques for further work.

▶ **Example 6.** Consider again Figure 2 and assume that `flag == true` indicates an *error-mode*. Changing the initial value of the variable `flag` from uninitialized to `false` changes the feasible WCET path of the program, and hence the WCET. In the such modified program the approach of Section 4(iii) infers that the loop is executed 5 times instead of 9 times as computed in Section 4. Applying the technique of Section 4(iii) to the modified program incrementally changes the ILP to reflect the change in the program behavior, resulting in a WCET for the new WCET path, that is a WCET when the program is not executed in the *error* mode.

Summarizing, similarly to Section 4 the three applications discussed in this section selectively apply symbolic execution to the program. In (i), we argue that bookkeeping the exact frequencies of program blocks can be done in a cheap way and exhaustive symbolic execution can be applied to derive loop bounds. In (ii) and (iii) we rely on the implemented symbolic execution infrastructure and use it in conjunction with the approach of Section 4(iii). This way, we only apply symbolic execution on a (reduced) number of WCET trace candidates, and avoid the burden of exploring all program paths.

6 Related Work

Symbolic execution was originally used for test-case generation and has recently found more and more applications in program verification, for example, in bug-hunting [6]. Applications of symbolic execution in program verification use various heuristics to speed up symbolic execution, identify and track relevant program information and use constraint solvers to prove caching queries. Our symbolic execution engine in r-TuBound offers only few heuristics and derives as much program information as needed for the WCET analysis. In general, inferring precise program information comes with high computational costs, a problem which we avoid by using selective symbolic execution: r-TuBound applies symbolic execution only when information about the program is too coarse or when other analysis methods fail.

A similar idea is presented in [4] where symbolic execution is used to refine spurious def-use results via a path feasibility analysis. In [4] branching decisions are determined at compile time and used to identify and remove infeasible paths. This method can be seen as a light-weight on-demand symbolic execution of conditional nodes, whereas symbolic execution in r-TuBound always executes single paths.

Symbolic execution for WCET analysis is also used in [13] and avoids some typical pitfalls of symbolic execution. For example, loops are not unfolded and hence multiple executions

of the same block are omitted. We note that [13] analyses each program block whereas our selective symbolic execution approach in r-TuBound only analyses relevant program blocks and paths.

A related approach is the abstract execution framework of [11], where context-sensitive abstract interpretation is applied to analyse loop iterations and function calls in separation. Instead of applying a fix-point analysis, abstract operations on abstract values are applied in [11], where an abstract value can, e.g. , be represented as an interval. When abstract values prevent the evaluation of a conditional, both branches need to be followed. Abstract states can be merged at join points to prevent the path explosion problem. As a result, a single abstract execution can represent execution of multiple concrete paths. This is not the case in the traditional use of symbolic execution. Compared to r-TuBound, abstract execution in [11] analyses the entire program, whereas in r-TuBound we apply symbolic execution only to relevant parts of the program. An integration of abstract execution in r-TuBound is an interesting research direction for future work.

In the traditional use of static WCET analyzers, high-level tools gather flow fact information about the program under analysis. This information is subsequently used in further (low-level) WCET analysis. Static WCET analyzers, see e.g. [16, 1, 9], often use the IPET technique [19] to calculate WCET estimates. This leads to an over-estimation of the WCET since the IPET modeling of a program usually encodes spurious execution traces that are infeasible in the concrete program. The approach of [17] addresses the problem of refining imprecise WCET estimates, by using symbolic execution in conjunction with IPET. We implemented this combination in r-TuBound. The results and applications of our implementation offer an automated technique to reduce, and possibly avoid over-estimation in WCET computation. A similar method is presented in [12], where an ILP encoding of the program is used to check whether partial solutions of a specific size to the ILP problem yield infeasible program paths. Feasibility of solutions is checked using model checking, by encoding block execution frequencies as program assertions. Unlike [12], we apply path-wise symbolic execution to avoid model checking the entire program and use SMT solving for checking feasibility of program paths.

Measurement-based timing analysis techniques, such as [20], can be seen complementary to static WCET analysis tools. Measurement-based tools require test inputs that cover a sufficient portion of the program executions to infer a tight WCET bound with a high confidence. The method of [20] systematically generates test-cases for arbitrary program executions, based on model checking and various heuristics. In the proposed application of symbolic execution in r-TuBound we generate test-cases only for program executions along the WCET trace candidate path(s).

A different approach to WCET analysis is given in [7], that relies on segment- and state-based abstract interpretation [8]. This state-based approach has similarities with the ILP problem refinement of r-TuBound. Integrating this approach in r-TuBound is an interesting task to be investigated.

7 Conclusion

We outlined applications of symbolic execution in WCET analysis, as implemented in r-TuBound: reasoning about single statements in loops, computing loop bounds, and refining the results of an a-priori used WCET analyzer. The approaches have successfully been tested on a number of WCET benchmarks. Additional applications of symbolic execution can be implemented in r-TuBound by only minor changes of the underlying symbolic execution

engine. With such changes at hand, we are confident that symbolic execution can also be used in hardware-aware dynamic WCET analyzers. We believe that, an efficient use of symbolic execution, called selective symbolic execution in this article, gives a valuable extension to the program analysis toolbox applied in WCET analysis.

References

1. C. Ballabriga, H. C. Hugues, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop (SEUS)*, pages 35–46, 2010.
2. M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve 5.5, Open source (Mixed-Integer) Linear Programming system. Software, 2004. `http://lpsolve.sourceforge.net/5.5/`.
3. A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. SmacC: A Retargetable Symbolic Execution Engine. In *Proc. of ATVA*, 2013. To appear.
4. R. Bodík, R. Gupta, and M. L. Soffa. Refining Data Flow Information Using Infeasible Paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, Nov. 1997.
5. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of TACAS*, pages 174–177, 2009.
6. C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, 2013.
7. P. Cerny, T. Henzinger, and A. Radhakrishna. Quantitative Abstraction Refinement. In *Proc. of POPL*, pages 115–128, 2013.
8. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*, pages 238–252, 1977.
9. J. Gustafsson. SWEET: SWEdish Execution Time tool. Software, 2001. `http://www.mrtc.mdh.se/projects/wcet/sweet/index.html`.
10. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present and Future. In *Proc. of WCET*, pages 136–146, 2010.
11. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS*, pages 57–66, 2006.
12. Ho Jung Bang and Tai Hyo Kim and Sung Deok Cha. An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation. In *Proc. of ISORC*, pages 365–372, 2007.
13. D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In *Proc. of WCET*, 2006.
14. R. Kirner. The WCET Analysis Tool CalcWcet167. In *Proc. of ISoLA (2)*, pages 158–172, 2012.
15. J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Proc. of PSI*, pages 116 – 126, 2011.
16. J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In *Proc. of LPAR*, pages 435 – 444, 2012.
17. J. Knoop, L. Kovács, and J. Zwirchmayr. WCET Squeezing: On-demand Feasibility Refinement. Technical Report 2013-V-1, TU Vienna, Institute of Computer Languages, May 2013.
18. A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proc. of WCET*, 2008.
19. P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, 1997.
20. M. Zolda and R. Kirner. Compiler Support for Measurement-based Timing Analysis. In *Proc. of WCET*, pages 62–71, 2011.

Upper-bounding Program Execution Time with Extreme Value Theory

Francisco J. Cazorla[1,2], Tullio Vardanega[3], Eduardo Quiñones[1], and Jaume Abella[1]

1 **Barcelona Supercomputing Center**
2 **Spanish National Research Council (IIIA-CSIC)**
3 **University of Padova**

Abstract

In this paper we discuss the limitations of and the precautions to account for when using Extreme Value Theory (EVT) to compute upper bounds to the execution time of programs. We analyse the requirements placed by EVT on the observations to be made of the events of interest, and the conditions that render safe the computations of execution time upper bounds. We also study the requirements that a recent EVT-based timing analysis technique, Measurement-Based Probabilistic Timing Analysis (MBPTA), introduces, besides those imposed by EVT, on the computing system under analysis to increase the trustworthiness of the upper bounds that it computes.

1998 ACM Subject Classification D.2.4 Software Engineering: Software/Program Verification

Keywords and phrases WCET, Extreme Value Theory, Probabilistic, Deterministic

Digital Object Identifier 10.4230/OASIcs.WCET.2013.64

1 Introduction

Extreme Value Theory (EVT) can be regarded as the counterpart of Central Limit Theory [7]: where the latter studies the bulk of the population of a given distribution, EVT studies the tail of it, in other words the extreme deviations from the median of probability distributions. By analysing a sample of observations of a given random variable, EVT determines the probability of extreme events to occur, where "extreme" refers to either end of the range of the value domain of those events. EVT is widely used in many disciplines, ranging from structural engineering to Earth sciences.

EVT has also been used to provide estimates of average-case execution time [17][16] and Worst-Case Execution Time (WCET) of software programs [8][5], which is the focus of this paper. In contrast to classic static WCET analysis, EVT computes a cumulative distribution function, or probabilistic WCET (pWCET) function, that upper-bounds the execution time of the program, guaranteeing that it only exceeds the given bound with a probability lower than some threshold (e.g., 10^{-15} per run).

EVT is applied in for measurement based timing analysis (MBTA). In MBTA, execution time measurements of the timing behaviour of the program of interest are processed by specialised EVT-based analysis to generate pWCET bounds for the program that should hold at deployment time.

In order for sound results to be obtained from the application of EVT it must hold that the observations of the events of interest can be regarded as random variables that are independent and identically distributed (i.i.d.). When this property cannot be asserted a-priori, it can be verified a-posteriori by suitable statistical tests [7]. However, EVT has

© Francisco J. Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 64–76

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

nothing to say on the *representativeness* of those data, that is, on the safeness of the pWCET estimate that is computed from them. Representativeness is determined by the quality of the data passed to EVT, or analogously, by relevant properties of the environment that generated those data. The pWCET estimates obtained with EVT are therefore valid only for the data population sampled for the analysis or, by extension, for the operating conditions subsumed by those data.

If representativeness is *low*, the pWCET bounds obtained with EVT do *not* provide any trustworthy prediction on the timing behaviour of the program on the target platform, but rather a description of the extreme timing behaviour that the program might have in the execution conditions exercised during the observation runs. If representativeness is high instead, more general conclusions can be drawn on the computed pWCET when the program is run on the target platform under execution conditions beyond those exercised during the analysis. The primary goal of MBTA techniques is to provide pWCET estimates that hold under execution conditions that may occur during the operation of the system: whereas those conditions may not be exactly identical to those captured by the observation runs made at analysis time, they should still represent them probabilistically. How this critical property can be asserted is the subject of this paper. This ability solves one of the key problems that real-time system designers have in trying to determine the timing behaviour of a real-time system. In this regard we show the benefits that can be achieved in terms of representativeness when using a time-randomised execution platform like the one proposed in the PROARTIS project [3][10][13], in contrast to conventional time-deterministic architectures[1].

In this paper we discuss the requirements on the use of EVT in computing pWCET bounds and the representativeness of the pWCET estimates. We show that increasing representativeness requires: (1) controlling the execution conditions at analysis time; and (2) understanding the representativeness of the analysis-time execution conditions with respect to those that may occur during operation. We discuss how *measurement-based probabilistic timing analysis* (MBPTA) based on EVT [5] reduces the burden placed on the user of the method for controlling the execution conditions. To that end, MBPTA requires that the effects that can be exercised by the execution conditions on the observation runs made during analysis are: either (1) bounded from above so that they represent worst-case effects; or (2) time-randomised; or else (3) ensured to have exactly the same probabilistic distribution at analysis *and* at deployment.

Contribution: This paper establishes the principles on which EVT can be used to derive WCET estimates in time-deterministic and time-randomised architectures. In particular, it helps WCET analysis community better understand the requirements, limitations and benefits that EVT carries on the determination of pWCET bounds, also understanding the requirements that MBPTA adds on top of EVT. This will set the baseline for future works in this promising area of research.

2 An executive introduction to EVT

EVT provides a canonical theory for the (limit) distribution of normalised maxima of independent, identically distributed random variables. EVT involves non-parametric statistics,

[1] A system is time-deterministic when we can determine its state at any time t on the basis of the initial state, inputs and the time cost of the state transition triggered by those inputs (read more on this in Section 4).

that is, EVT does not rely on data (i.e. the population under study) belonging to any particular distribution. EVT describes the behaviour of extremal events for stochastic processes that evolve dynamically in time and space. It gives the user a methodology for predicting the occurrence of rare events. Estimating distribution tails beyond the limit of available data is a complex process that requires making mathematical assumptions on the tail model. These assumptions are very difficult if at all possible to verify in practice. There is thus intrinsic risk in choosing the tail model to fit the problem at hand, which is necessary to correctly apply EVT.

We are interested in the high values that bound the pWCET, so we consider the EVT prediction for maximal values of a set of observations. There are two main approaches to EVT. The first, known as Peak over Threshold method [2], models a distribution of excess over a given threshold: EVT shows that the limiting distribution of exceedance is a Generalised Pareto Distribution or GPD. The second approach, which we use, known as Block Maxima model (BMM) [7] considers the largest (smallest) observations obtained from successive *periods* (blocks), where the selection of the block size is a critical parameter. Under BMM the asymptotic distribution of the maxima (minima) is modelled and the distribution of the standardised maxima is shown to follow one of the Gumbel, Frechet or Weibull distributions [7]. The generalised extreme value distribution (GEV) is a standard form of these three distributions.

3 Requirements on the use of EVT for WCET estimation

When used to predict the extreme (hence worst-case) timing behaviour of applications running on a computing system (platform), EVT is given in input a number of *observations* taken from real execution of the system of interest. In our case, these observations are execution time measurements from runs of the program of interest, taken, under controlled execution conditions (hence, during analysis) on the target platform. From these observations EVT infers an approximation of the tail of the timing behaviour of the program that hold during actual operation.

EVT requires observations coming from the system under study to be described mathematically as random variables that are i.i.d. [7]: Two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution function. In our application of EVT, identical distribution holds when those two random variables describe the same system using the same set of parameters in the same way (whether deterministically or probabilistically), for all inputs with influence on the timing behaviour of the program, including input vectors and initial hardware and software state.

EVT does not describe how the input observations are made: it regards the system as a black box of which it is only interested in considering the external manifestations (observation of runs here) which have to be i.i.d. for theory to apply, see Figure 1(a). Observations evidently describe *some* behaviour of the system: however, as argued in this paper, it must also be ensured that the execution conditions under which those observations are taken at analysis time, do represent the execution conditions that will occur at deployment time, for which the computed pWCET estimates are required to provide a safe upper bound. This requirement imposes significant overhead on the user in: (1) understanding the execution conditions that the programs of interest may experience at deployment time; and (2) controlling the execution conditions during analysis time so that they are significantly representative. If not

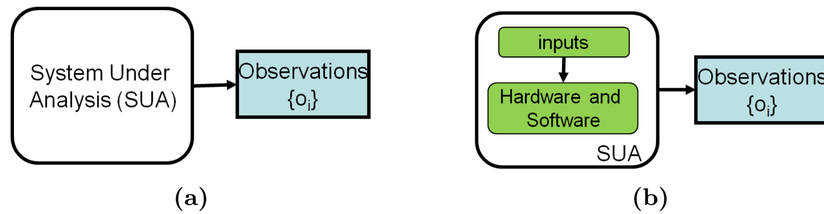


Figure 1 (a) system under analysis as assumed by EVT; (b) a computing system for analysis with EVT.

fulfilled properly, these two obligations can make the whole approach of deriving estimates based on EVT utterly misleading.

3.1 Defining the population under analysis

One of the most critical steps in applying EVT is understanding and capturing the population (universe) to be analysed, including the features of that population that are relevant for the analysis. In our case, the population is given by all the runs that the program of interest will perform at deployment time, and the feature of interest is the execution time of those runs. For instance, considering a program to be deployed on an aircraft, each run of that program in it during its whole operational lifetime is an individual of the population of interest. If the aircraft had a lifetime of 25 years, flying 80% of that time, and the program under analysis had an execution period of 100 ms (i.e., 10 times per second), the total population would be comprised of nearly 63×10^8 elements.

In general, the total population of events in a real-world system is inordinately large and hard to determine, so it cannot be fully enumerated at analysis time. User intervention is therefore needed to cap the population of interest to a treatable dimension, in a trustworthy, timely and cost-effective manner. This step requires understanding the sources of influence on the feature of interest of the population, which means understanding what factors in the system affect the execution time of any given run of the programs under study. We call those factors: *sources of execution time variability*.

3.1.1 Sources of execution time variability

The execution time of a program is affected by several main factors, namely, the input conditions and the internal logic (state-dependent behaviour) of the platform (i.e. its hardware and software resources underneath the application layer), see Figure 1(b). Those factors represent what we call the Sources of Execution Time Variability ($SETV$) of the program. Each combination of values in the $SETV$ defines one particular *execution condition* under which a given run of the program may occur.

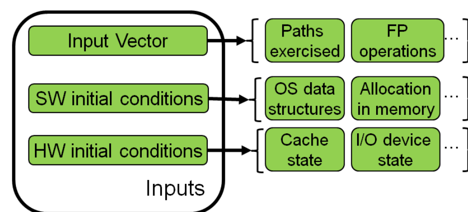


Figure 2 Main input components.

In our discussion here "input conditions" are understood to refer to 3 main components, see Figure 2: the input vector; the initial state of the software (for the program and the operating system, OS, underneath it); and the initial state of the hardware. Let us look at each of them in isolation.

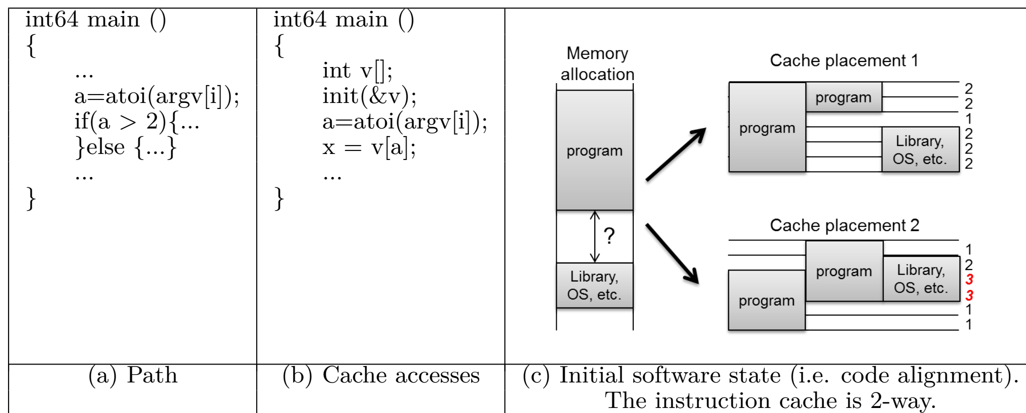


Figure 3 Examples of sources of execution time variability: programs for which the input vector affects: (a) the execution path; (b) the cache access pattern; (c) the cache jitter caused by the initial program state.

Input vector is the complete description of all the input data passed to the program which may affect the program execution behaviour. Input vectors may affect the hardware and the software state in all resources that are sensitive to history of execution. At software level, the most evident manifestation of the influence of input vectors is the execution path taken by the program during a given run. This is shown in Figure 3(a) where a is an element of the input vector. At hardware level, we find a variety of cases in which input vectors affect the timing behaviour of some components: Figure 3(b) shows the case where the memory address of a cache access (whose outcome may thus be a hit or a miss) depends on an input value.

Software initial state includes the initial state of all the read-write (state-sensitive) data structures used by the program directly or indirectly, the latter being those used by the operating system underneath it. It also includes all external aspects of the program that may have an influence on its execution-time behaviour: with processor resources such as the cache, the location in memory of the program determines the cache placement and the consequent cache conflicts of each memory access. Figure 3(c) shows a program using some external software components such as libraries or OS structures. Those components are allocated independently in memory and therefore, their relative alignment in cache can vary. As shown, different memory allocations (left) may lead to different cache placements (right). We show two particular cache placements: in the first one (top right) no cache set requires more than two lines to store all code, whereas in the second one (bottom right) two cache sets require up to 3 cache lines. Hence, if the instruction cache is 2-way set-associative, the former allocation is likely to produce better performance than the latter.

Hardware initial state includes the initial state of all the hardware resources (e.g., cache state) used by the program or any other software invoked by the program. Those states are important as the operation logic of many hardware and software components is state-dependent. The authors of [12] show that the initial state of the cache can be exploited to decrease WCET bounds by considering how cache contents may survive across subsequent disjoint executions of the program of interest.

3.1.2 Max population

We mentioned earlier that the execution time of a program is affected by several sources of execution time variability: $\{SETV_1, SETV_2, ..., SETV_j, ...SETV_n\}$, where source $j \in \{1, ..n\}$ can take up to k distinct values, that is: $SETV_j = \{v^j\}_k = \{v_1^j, v_2^j, ...v_k^j\}$. Consider for instance a multiplication unit whose response latency depends on its operands: the $SETV$ for this resource can take 2^m values, where m is the number of multiplications that occur in the program.

The sources of variability include all system inputs (input vectors, and SW and HW initial conditions)[2]. Controlling all the $SETV$ for each individual (i.e. measurement run) in the population is unfeasible as the target population is given by all the runs of all calls of the program under study that occur during operation.

The user has to derive a *population of maxima* (*max population*) that provides a safe upper bound of the target population, see the top part of Figure 4(a). If that is granted, then the execution times (i.e. individuals in the max population) safely upper bound the execution times in the actual population. In principle this requires analysing in detail each $SETV_j, \forall j \in \{1, ..n\}$. For each $SETV_j = \{v^j\}_k$ the user has to derive a subset of values $max(SETV_j) = max(\{v_1^j, v_2^j, ...v_k^j\}) = \{mv_1^j, mv_2^j, ...mv_{mk}^j\}) = \{mv^j\}_{mk}$. Eventually, based on the simplifying assumption of independence, we would need $max(SETV_j)$ to contain only its worst-case values. Identifying them may be so complex, however, that the user may have to resort to safe upper bounds. Moreover, to make the problem tractable, $mk < k$ should also hold as using $max(SETV_j)$ in place of $SETV_j$ serves the purpose of decreasing the size of the population of interest.

The actual cardinality of the max population is defined by the number of combinations of the max values that each $SETV$ can take during operation. If all $SETV$ are independent of one another, there is an individual in the max population for each element in the Cartesian product of all max values of all $SETV$: $\{mv^1\}_{mk} \times \{mv^2\}_{ml}, \times ... \times \{mv^n\}_{mm}$. Therefore, there is one execution time individual in the max population for a run under the execution environment defined by each of the combinations of all $SETV$. Otherwise, if some $SETV$ are not independent, then unfeasible combinations should be removed from the Cartesian product of all max values of all $SETV$. How to identify those combinations may be very hard.

3.2 Achieving i.i.d. behaviour

Ensuring that the observations submitted to EVT fulfil the required i.i.d. properties is only possible if the whole system behaves as a *random process* such that the observations drawn from it exhibit those properties. Given that the execution times of the programs under study are affected either by the input vectors passed to them or the internal logic (state-dependent behaviour) of the platform, the required i.i.d. properties on the observations drawn from the computing system must be obtained by: either (1) applying a random process on the way inputs are selected; or (2) applying random processes in the internal behaviour of the hardware/software (the platform). In both cases, WCET estimates are based on measurements (observations) taken from the execution of the program under study on the target platform. Hence, both approaches can be regarded as *measurement-based* rather than *static* as it is the case with other timing analysis techniques [18].

[2] For the purposes of this paper we assume $SETV$ to operate independently of one another. Later in this section we show how this simplifying assumption can be removed.

Next we discuss each of the two approaches in more detail, paying special attention to the requirements they place on the user as well as on the execution platform itself.

4 Deriving WCET estimates on time-deterministic systems with EVT

Current computing systems can be regarded as *time-deterministic*. Time-determinism is achieved when we can determine the state of the system at any time t on the basis of the initial state, the inputs and the time cost of the state transitions triggered by those inputs. The property of time determinism is disjoint from that of *functional determinism* in that a functional deterministic system can also be non time-deterministic: consider for example a system whose functionality can be fully described by a finite state machine. Further assume that the transition time from any two states *S0* to *S1* is a random value in the range [t1:t2]. This system is functionally deterministic in that it will always finish in state *S1* from state *S0*, but it is not time-deterministic because we cannot determine the exact time at which the transition from *S0* to *S1* will complete.

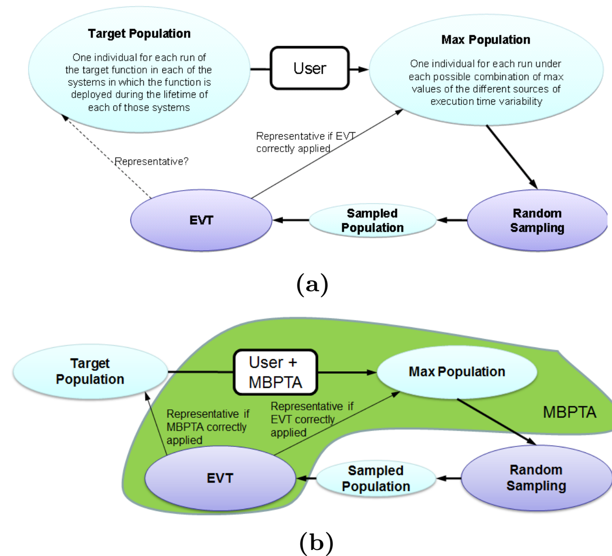


Figure 4 Application of EVT (top) and MBPTA on a time-randomised platform (down).

More specifically to our discussion, the execution time of a program on a time-deterministic system is constant across different runs that occur under the same execution conditions, i.e. the initial state in the hardware and software is fixed and the same input data are used across runs.

The behaviour of the system (the program running on the computing system) should behave like a *random process*, X_n, such that the i.i.d. statistical properties required for EVT can be had. If EVT is applied to a COTS (commercial off-the-shelf) deterministic computing system, the system cannot be changed for it to behave as X_n. In that case EVT conformance can only be sought by operating on the inputs submitted to it, as captured in Figure 2, in particular by randomising the selection of inputs using *random sampling*.

4.1 Random sampling

A sample is a set of individuals chosen from the population under study. A *random sample* is a sample chosen by random sampling from the population.

Several unbiased sampling methods exist, e.g, simple random sampling (SRS) [4]. An unbiased random selection of individuals is important so that, in the long run, the sample has the same statistical properties as the population under study. With SRS, each individual is chosen randomly and entirely by chance, hence providing independence between each picking of an individual. Further, each individual has the same probability of being chosen and each

subset of k individuals has the same probability of being chosen as any other subset of k individuals. Hence, by construction, SRS constructs i.i.d. random samples.

However, as shown earlier, the target population under study cannot be fully constructed in the general case, and therefore it cannot be sampled. The max population is sampled instead. In our scenario this has an important corollary in the dimension of *representativeness of the WCET estimates computed with EVT*. If and only if the max population is a safe upper bound of the (deployment-time) target population, the WCET estimates that can be obtained with EVT actually upper bound the timing behaviour of the program of interest during operation. Hence, a critical step to achieve the required representativeness is to understand the deployment-phase target population, correctly defining the max population. Failing to do so would cause the WCET estimates obtained with EVT to lack any statistical representativeness with respect to the deployment-phase population. Under that scenario, the testing-time behaviour observed cannot be used by EVT to derive WCET estimates of the deployment-time behaviour of the program on the target system.

4.2 Deriving a safe max population

This step is one of the most complex passages in the procedure of applying EVT to derive pWCET estimates that hold at deployment time for the timing behaviour of the programs of interest. This step requires analysing in detail each of the $SETV_j$, ensuring that all the components of $max(SETV_j)$ take their respective maximum value(s), or value(s) that upper-bound their maximum. Indeed, it must be the case that the individuals leading to the pWCET must be part of the combination of $max(SETV_j)$ values or the upper bounds thereof. This condition holds if no timing anomalies occur across $SETV$, so that the combination of local worst cases in $max(SETV_j)$ leads to the global worst case. Some SETV are well understood by the user who can then control and bound them, either quantitatively or qualitatively, as we discuss below. Others are harder to enumerate and consequently difficult to bound from above.

Qualitatively-boundable SETV. The input vectors are the most challenging case of SETV, for the effect they have on the execution paths taken by the program. A poor path coverage may cause the results computed by EVT to miss the safeness quality required for worst-case timing analysis. To use EVT the user should understand what the worst-case paths in the program are and providing input vectors that exercise them. And the pWCET estimates derived with EVT would only be valid for the paths observed during analysis.

Tool support may be available to compute the coverage obtained during observation runs and report it back to the user. If a path that the tool deems possible is not yet covered in the observation runs, the tool may request that the user either acknowledges the exclusion of that path (which may be asserted as irrelevant for WCET estimation) or provides further test cases to exercise that path. This is technically possible because full path coverage at source level (possibly reduced to MC/DC [9]) is one of the prerequisites to be satisfied by functional testing in certified systems. This type of timing analysis should thus be performed in conjunction with the functional verification campaign.

Quantitatively-boundable SETV. The input vectors do not only affect the execution path followed in a program, but may also influence other resources with jitter, hence creating another $SETV$, causing an inordinate increase in their quantity. Returning to the multiplier whose response latency depends on its input values, the user should provide input vectors for each program path in which the distribution of multiplier input values upper-bounds

the one that can occur during operation. If for example the multiplier takes 1 cycle when one operand is zero and 4 cycles otherwise, then the user is required to ensure that the distribution of non-zero values in the analysis input vectors upper bounds the distribution that may occur upon deployment.

Uncontrollable SETV. Controlling all $SETV$ to a level in which all their values can be known and a subset of them (max values) can be forced to occur in the computing system is generally out of reach for the user. Assume for example that the only $SETV$ is the location in memory where objects are placed (e.g., the program data and instructions, libraries, OS data and instructions, etc.) since this determines cache behaviour. In this scenario, enumerating all combinations of object placements in memory is simply unfeasible. As shown in [11], the number of memory alignments leading to different cache placements is $s^{n_{obj}-1}$, where s is the number of cache sets and n_{obj} the number of memory objects. Moreover, the user has limited control to force a given alignment. It therefore follows that some $SETV$ are hard to define, understand and control by the user.

4.3 Summary

EVT requires that the observations taken during analysis warrant independence and identical distribution. For a population of maximum values on which SRS is applied, independence can be preserved by controlling how experiments are made. This requires ensuring that: (1) no source of dependence can exist between end-to-end runs; and (2) no state-dependent effect occurs in the processor *and* no logical software-level state is allowed to pass between any two runs. Whether that dependence may exist across events or instructions within a run is irrelevant so long as observations are collected at the granularity of end-to-end runs.

The fact that each element (an individual in the max population) has the same probability of being chosen and each subset of k individuals has the same probability of being chosen for the sample as any other subset of k individuals, ensures identical distribution. However, EVT by itself does not ensure the representativeness of the max population with respect to the *actual* population, which may not be fully known by the user and hence not be sampled. EVT makes no claim on how safe the selected "maximum population" is for the purposes of upper-bounding the actual population of system events of interest.

Ensuring that the computed pWCET estimates are safe upper bounds of the target population requires controlling all $SETV$. Our view here is that attempting to provide WCET estimates on COTS deterministic systems is fatally limited by the intricate dependences of the $SETV$ and the hardware/software support to control $SETV$ to a level in which all their values can be known, the maximum can be identified and forced in the computing system to carry out pertinent runs to feed EVT. There is therefore a risk of taking as valid for the program EVT projections whose representativeness cannot be assessed beyond the particular set of inputs used during the analysis. To apply EVT in MBTA, therefore, the user must be provided with means to derive max population safely, timely and cost-effectively.

5 Deriving WCET estimates on time-randomised systems with EVT

In order to ensure that the computed pWCET estimates are safe upper bounds to the execution time of the program at deployment time, MBPTA adds further constraints to those imposed by EVT [5], see Figure 4(b). While the EVT requirements only concern the nature of the observations, MBPTA requires controlling the inputs submitted to the program

and the platform on which the runs occur, as shown in Figure 1(b). In particular, MBPTA requires that all the $SETV$ are "controlled" in one of the following ways.

1. **Safe upper-bounding of $SETV$ with no HW change.** The user provides values for each and every $SETV$ to upper-bound the effect that those $SETV$ can have on the execution time of the program during operation. For instance, for "reasonable" replacement policies, an empty initial cache state represents the worst-case state that a program may find at start up. Similarly, for path-dependent effects, the user needs to ensure that the paths of interest to pWCET estimation are traversed during the observation runs.
2. **Removal of $SETV$ with HW change.** Some $SETV$ are hard, if at all possible, to be effectively controlled by the user. For those $SETV$, the processor hardware, and to a lesser extent, the software, should be redesigned such that the response time jitter of the corresponding execution resources does not depend anymore on the relevant $SETV$. This approach is tantamount to removing the $SETV$ for those resources. We consider two ways in which this can be had[3]:
   a. **Worst case timing behaviour.** At software level this approach consists in forcing relevant software components (e.g., methods, procedures) to always take the same time to execute regardless of when they are called. This approach has been followed for OS calls [1], where the jittery part of the required activity is deferred until after the return from the call, in the interstices between the execution of application programs, taking care to not incur disturbing perturbations to hardware state left on return. At hardware level, the worst-case mode [15] is a feature forcing a hardware block to take its worst delay even if a particular request finishes earlier. Both features make the observations obtained at analysis time be a safe upper bound of the deploy-time behaviour of the program.
   b. **Time randomisation.** Forcing the worst-case response of some hardware/software components to occur at all times may degrade performance significantly (e.g., considering all cache accesses as misses). Instead, randomising the timing behaviour of a given resource, significantly improves performance of that resource and makes observations taken at analysis time to be representative of the deployment time behaviour. If enough observations of the execution of that resource are taken, the observed frequencies converge to the actual probabilities. Further, randomisation may help removing some of the $SETV$ as shown in coming section.

5.1 How to achieve time randomisation

One of the main challenges in the context of MBPTA is understanding whether timing may be randomised for some hardware resources, while of course leaving functional behaviour unaffected. Time randomisation removes some of the $SETV$ that affect the execution time of programs, thereby reducing the burden on the user in applying EVT. For the sake of illustration we focus on the case of the cache, though the principles we present apply to any other resource.

We have seen earlier that the memory layout of program data and code is a $SETV$ that affects the program execution time. It is well known that the location in which program

[3] A third way exists if the user can provide inputs that have the same distribution of values, with respect to a given characteristic under analysis, as those that can appear during operation. This is possible but exceedingly difficult.

data and code are placed in memory may vary across runs (or upon composition with other software). This in turn means that the particular sets in which the different data/instructions are located vary across runs. An important consequence of that phenomenon is that the conflicts in cache that a program suffers and their effect on execution time may vary across runs. This is illustrated in the example in Figure 3(c), which shows the case of the code placement in memory and its effect on the instruction cache.

In a real-world scenario, expecting the user to control the way in which program data and code are placed in memory is not practical: tools and methods exist to do so [14], but they place some burden on the user. MBPTA can be facilitated by randomising the placement policy, and optionally the replacement policy (which is not covered in this discussion), to cause the effect of placement and replacement policies to become probabilistically analysable. The basic idea is to break the causal dependence between the particular address in memory in which a piece of data is, and the particular cache set in which it is allocated. This dependence is broken by *randomising the placement* such that the index set of individual data items is randomised and made vary across runs. In this way, making enough runs, hence taking enough observations, is sufficient for the user to provide probabilistic evidence of the effect of placement on execution time. And more importantly, in each run the user needs not control the particular location in memory in which individual data items are located. Random placement can be done at hardware level by changing the design of the cache [10] or at software level by controlling the way data and code are loaded in memory [11].

Overall, in a MBPTA-compliant platform (i.e. one that conforms to the principles we illustrated) all $SETV$ are under control, whereby: (1) their effect on the execution time observations taken from the program is known; (2) the representativeness of the observed execution times of the program as affected by those $SETV$ at analysis time is guaranteed with respect to the execution times of the program at deployment time; and (3) the observed execution times can be regarded as independent and identically distributed random variables so that EVT can be meaningfully applied.

6 Related work

Extreme-value statistics are used in [6] to model the WCET. To select highest execution times the cited authors use Block Maxima [7], for which instead the authors of [8] use the Peak Over Threshold method. In [19] the authors apply EVT to the problem of computing Worst-Case Response-Time (WCRT) of programs. Older papers focus on the application of EVT on time-deterministic platforms but do not cover the representativeness of the result obtained from EVT, that is the safeness of the pWCET estimate. The authors of those works assume that the observations collected are representative of the *target population*. In this paper we have shown the difficulties of achieving representativeness in time-deterministic platforms.

The authors of [5] present MBPTA as well as the requirements that it imposes on the hardware and software components of the system to increase the trustworthiness of the upper bounds computed by MBPTA, primarily the time randomisation of certain processor resources.

7 Conclusions

While EVT has been regarded as a powerful method to derive upper (lower) bounds on arbitrary distributions, its utilisation for deriving WCET estimates has not been well

described yet. In this paper we review those characteristics of WCET estimation relevant for the use of EVT. In particular, we present how the initial population for EVT should be generated, including all sources of execution time variability (SETV), and how this process can be hopeless on time-deterministic platforms. Conversely, we show that time-randomised platforms enable an effective use of EVT by means of MBPTA by randomising and upper-bounding the timing behaviour of some hardware and software resources so that some SETV do not need to be described by the user while WCET estimates obtained are still sound.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROARTIS Project (www.proartis-project.eu), grant agreement no 249100. This work was partially supported by EU COST Action IC1202 "Timing Analysis On Code-Level (TACLe)", and by the Spanish Ministry of Science and Innovation under grant TIN2012-34557. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the Juan de la Cierva grant JCI2009-05455.

References

1. A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. *WCET Workshop*, 2012.
2. J. Beirlant, Y. Goegebeur, J. Segers, and J. Teugels. *Statistics of Extremes: Theory and Applications.* 2004.
3. F.J. Cazorla, E. Qui nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.
4. W.C. Cochran. *Sampling Techniques.* 1977.
5. L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
6. Edgar S and Burns A. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–225, 2001.
7. W. Feller. *An introduction to Probability Theory and Its Applications.* 1996.
8. J. Hansen, S Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
9. Hayhurst K.J., Veerhusen D.S., Chilenski J.J., and Rierson L.K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.
10. L. Kosmidis, J. Abella, E. Quinones, and F.J. Cazorla. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
11. L. Kosmidis, C. Curtsinger, E. Quinones, J. Abella, E. Berger, and F.J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.
12. L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, and F.J. Cazorla. Achieving timing composability with measurement-based probabilistic timing analysis. In *ISORC*, 2013.
13. E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. *WCET Workshop*, 2011.
14. E. Mezzetti and T. Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*, 2013.
15. M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

16 P. Radojković, P.M. Carpenter, M. Moretó, A. Ramirez, and F.J. Cazorla. Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem. In *MICRO*, 2012.

17 P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F.J. Cazorla, M. Nemirovsky, and M. Valero. Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *ASPLOS*, 2012.

18 Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

19 L. Yue, T. Nolte, I. Bate, and L. Cucu-Grosjean. A statistical response-time analysis of real-time embedded systems. In *the 33rd IEEE Real-time Systems Symposium*. IEEE, December 2012.

PRADA: Predictable Allocations by Deferred Actions*

Florian Haupenthal and Jörg Herter

Saarland University – Computer Science
Saarland, Germany
`{s9flhaup@stud, jherter@cs}.uni-saarland.de`

Abstract

Modern hard real-time systems still employ static memory management. However, dynamic storage allocation (DSA) can improve the flexibility and readability of programs as well as drastically shorten their development times. But allocators introduce unpredictability that makes deriving tight bounds on an application's worst-case execution time even more challenging. Especially their statically unpredictable influence on the cache, paired with zero knowledge about the cache set mapping of dynamically allocated objects leads to prohibitively large overestimations of execution times when dynamic memory allocation is employed. Recently, a cache-aware memory allocator, called `CAMA`, was proposed that gives strong guarantees about its cache influence and the cache set mapping of allocated objects. `CAMA` itself is rather complex due to its cache-aware implementations of split and merge operations.

This paper proposes `PRADA`, a lighter but less general dynamic memory allocator with equally strong guarantees about its influence on the cache. We compare the memory consumption of `PRADA` and `CAMA` for a small set of real-time applications as well as synthetical (de-) allocation sequences to investigate whether a simpler approach to cache awareness is still sufficient for the current generation of real-time applications.

1998 ACM Subject Classification B.3.3 Performance Analysis and Design Aids

Keywords and phrases Dynamic Memory Allocation, Worst-Case Execution-Time, Cache Predictability

Digital Object Identifier 10.4230/OASIcs.WCET.2013.77

1 Introduction

(Hard) real-time applications raise the requirements on dynamic memory allocators. Constant (de-) allocation times and a bounded, predictable cache behaviour become equally important as *good* response times and low memory consumption. Short, constant response times can be guaranteed by a large set of dynamic memory allocators, ranging from conventional buddy systems [9, 13] to specialized real-time allocators like Half-Fit [12] and `TLSF` [11]. However, none of these allocators provide guarantees about their effects on the cache that a cache analysis may exploit to provide a subsequent timing analysis with a tight approximation of the program's cache behaviour.

With `CAMA` [8, 6], the first cache-aware constant-time dynamic memory allocator was proposed. This allocator guarantees constant execution times as well as a bounded cache influence on just a statically known set of cache sets for allocations and deallocations.

* This work was supported by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

© Florian Haupenthal and Jörg Herter ;
licensed under Creative Commons License CC-BY

Furthermore, CAMA can be guided to which cache sets newly allocated objects shall be mapped. Common techniques to counteract memory fragmentation like splitting[1] and merging[2] are used. To implement these operations without introducing unpredictable cache effects, CAMA relies internally on an indirect management of free blocks. Internal free lists and split/merge operations work on so-called descriptors instead of the free blocks themselves. Strict memory placement policies exist for descriptors to ensure a statically predefined cache mapping of descriptors as well as the absence of accesses to unknown cache sets.

PRADA is a lighter implementation without the need for descriptors, providing the same predictability guarantees: constant execution times and a statically known effect on the cache. PRADA tackles the challenge of not introducing cache unpredictabilities by performing (partial) splits/merges only when no other cache set than the one already touched during the (de-) allocation procedure is accessed. To enable PRADA to choose when to perform (partial) split/merge operations, all these operations are initially deferred and executed when the prerequisites for the operation are met. However, the allocator does not provide any guarantee that these deferred actions will be executed at all. It only stores a fixed, but configurable amount of deferred actions. Surplus ones are simply dropped, i.e., never executed.

For general purpose dynamic memory allocators, deferred split and merge actions have been shown to be inferior to immediate splits and merges [14]. But does this still hold true when restricting the class of programs in which an allocator may be used to (hard) real-time applications? In this paper, we investigate on whether an implementation as complex as CAMA is actually necessary to fulfil the raised demands of hard real-time systems; or whether we can do with a simpler approach like PRADA.

In Section 2, we describe PRADA, an alternative cache-aware constant time dynamic memory allocator that uses deferred actions in order to implement split and merge operations in a (cache- and time-) predictable manner. Section 3 studies the memory consumption of several dynamic memory allocators when presented (de-) allocation sequences representative for real-time applications. Related work is summarized in Section 4.

2 PRADA

PRADA is a dynamic memory allocator which manages free *memory blocks* in segregated free lists to allow for constant time allocation and deallocation routines. It defers actions which would introduce unpredictable behaviour when always immediately executed by remembering them in form of *requests*. These actions are executed during subsequent (de-) allocations to the cache set they need to access. This section briefly summarizes how PRADA works and achieves its predictability goals. A more detailed description of PRADA can be found in [3].

The memory managed by PRADA is divided into memory blocks. Every memory block consists of a size field storing its current size and the actual payload area itself. PRADA uses the payload area of currently free blocks to build-up its free lists. Therefore, the payload area of deallocated memory blocks contains three fields. Two list pointers linking to the previous block and the next block within the free list, respectively. The third field is a pointer to a potentially pending request.

PRADA and CAMA use their respective free lists in the same way. They use an adapted

[1] Splitting denotes the use of (split parts of) larger blocks in order to satisfy a request for a smaller block.
[2] Merging denotes the joining of two physically consecutive free blocks in order to have a larger block with higher probability to be useful in satisfying an allocation request.

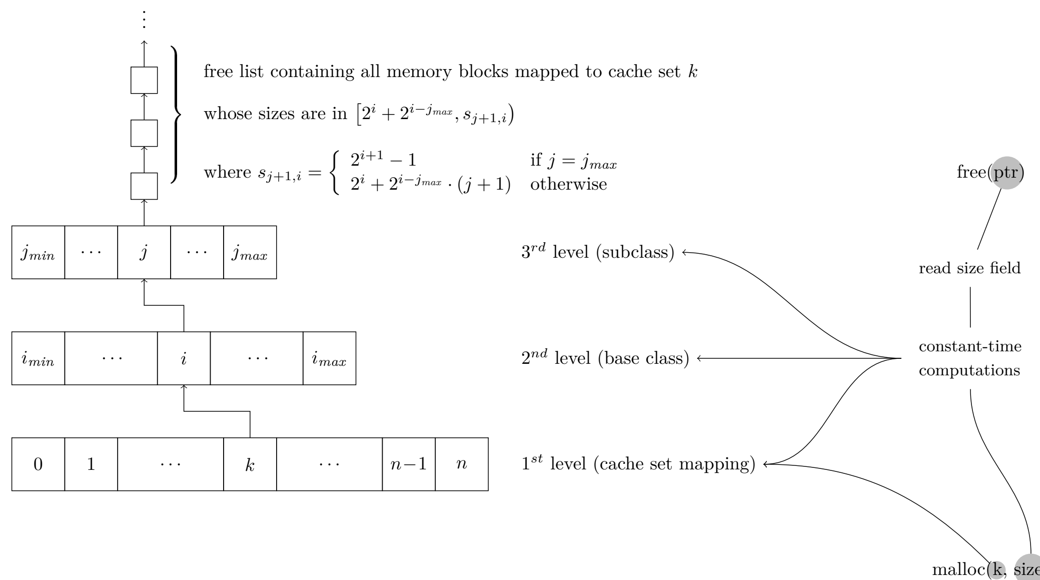


Figure 1 Logical view on the partitioning of the memory in PRADA and CAMA and how to connect (de-) allocation requests to free lists.

version of TLSF's two-level approach of building size classes (which in turn correspond to its free lists). During an allocation, the first block of a suitable free list, i.e. size class, is used to satisfy the request. How does the two-level approach to set-up size classes of TLSF, PRADA, and CAMA work? The (logically) first level sorts blocks in exponentially growing size classes, i.e., for class i all associated memory blocks are of $size \in [2^i, 2^{i+1})$. A higher granularity for sorting blocks is achieved by a second level which is a linear subdivision of these classes. For an allocation of size s, two computations are needed. First, the base class needs to be determined, then the correct subclass. Both classes can be computed in two constant-time computations:

$$class = \lfloor \log_2(s) \rfloor \text{ and } subclass = \left\lfloor \frac{(s - 2^{class}) \cdot j_{max}}{2^{class}} \right\rfloor$$

where j_{max} is the number of linear subclasses. CAMA and PRADA add an additional level to setting up free lists by firstly sorting free blocks according to the cache set they start in. In contrast to non-cache-aware allocators, they use an additional argument which allows to guide their allocations to a certain cache set. This additional argument selects which free list structure is searched. Figure 1 illustrates how (de-) allocation requests are mapped to a suitable free list.

PRADA defers split and merge actions to avoid unpredictable effects on the cache during allocations and deallocations. Therefore, actions need to be remembered. Remembering actions has to be done in a way which preserves the predictable behaviour of the allocator. Therefore, we use an array of fixed size which contains requests for actions. These requests are used for the deferred performing of splits and merges. For each cache set, there is the same, fixed amount of entries reserved. Since this array is statically allocated, the impact on the cache state of accesses to this array is known. Due to this static setting, requests are

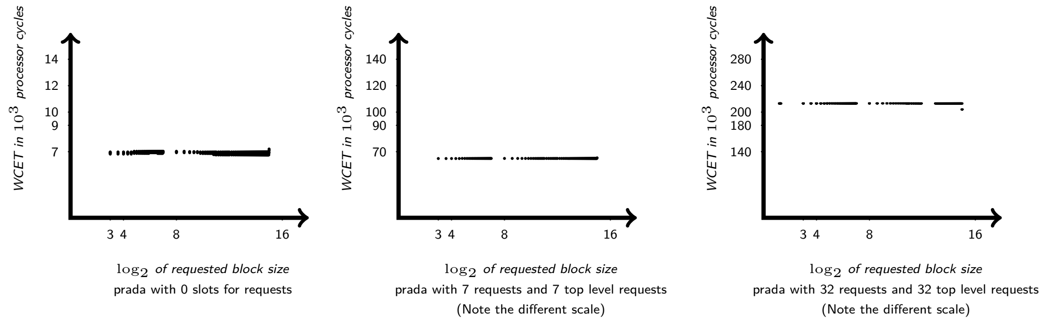


Figure 2 WCET bounds of `PRADA0`, `PRADA7`, and `PRADA32`.

never created or deleted, but get filled and cleared. These requests differ from the descriptors used by `CAMA` in two ways. The number of descriptors managed by `CAMA` depends on the number of managed memory blocks. For `PRADA`, the reserved space for requests is fixed. Furthermore, a request itself is smaller than a descriptor (8 bytes and 24 bytes, respectively).

`PRADA` executes one pending request, if there exists one for the current cache set during each allocation and deallocation. The following two paragraphs describe the procedures for allocations and deallocations and highlight when deferred requests are executed.

`PRADA` allocates memory blocks aligned to cache sets. Hence, all allocations (including the added space for the block header) are rounded up to the next multiple of a cache line. With the provided cache set mapping and the computed *class* and *subclass*, the free list containing the smallest blocks suitable to satisfy the allocation request is fully determined. If there exists a suitable block, i.e., the free list is not empty, there may also exist a pending, now obsolete merge request for this block. This request needs to be cleared first. Then, one pending request for the current cache set can be executed. If the selected block is exactly of the requested size, the block is simply returned. Otherwise, i.e. the found block is larger than requested, its size is set to the requested size and a split request is created. If no suitable block is found, new memory is requested from the operating system to create a suitable block, possibly requiring a deferred split operation. Even in case that no block was found, a requested action for the current cache set can be executed.

At deallocation, `PRADA` first checks whether there is still a pending split request for the deallocated block. If there is one, this split request is dropped and the blocks original size is restored. Then, one pending request for the current cache set is executed if one exists. Finally, the current block is inserted into the appropriate free list and marked as available for merges, i.e., a merge request is created.

Figure 2 depicts the WCET bounds for allocations of different sizes and mapped to different cache sets derived by aiT[4] for our prototype implementation of `PRADA`. For comparison, Figure 3 shows the respective WCET bounds derived for `TLSF` and `CAMA`. Deallocations take only a single pointer as an argument. For `PRADA`, the WCET bound for deallocations still depends on the number of requests. WCET bounds of 2,437 cycles, 59,947 cycles, and 182,783 cycles where derived for 0, 7, and 32 requests, respectively. For `TLSF` and `CAMA`, the bounds are 6,018 cycles and 98,156 cycles, respectively.

3 Evaluation

For the evaluation of `PRADA` and the comparison with `CAMA`, we used the relevant programs from the MiBench benchmark suite [2], i.e., those using dynamic memory allocation. The

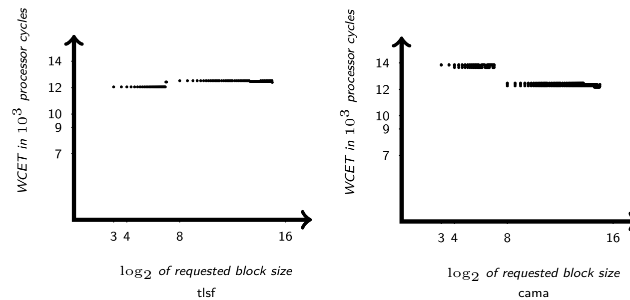


Figure 3 WCET bounds of TLSF and CAMA.

MiBench suite itself consists of a set of embedded programs, considered to be representative for commercial applications. However, most embedded systems avoid dynamic memory management. Therefore, we have only six relevant test cases from this suite. These six test cases run the programs `Susan`, `Patricia`, and `Dijkstra`, each on a set of small and large input data, respectively. `Susan` was developed for recognizing corners and edges in magnetic resonance images of the brain. The software is, however, also used as image recognition in unmanned vehicles. The small input data run processes a black and white image of a rectangle, while the large input data run processes a complex picture. A patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Patricia tries are often used to represent routing tables in network applications. `Patricia` uses patricia tries to construct a routing table. `Dijkstra` constructs a large graph (as an adjacency matrix) and then computes the shortest paths between pairs of nodes using repeated applications of Dijkstra's algorithm.

To get a better impression on their respective memory performances, we compare the total memory consumption of `CAMA` and `PRADA` against several other allocators:

- `TLSF`: a constant time, but cache-unaware *real-time* allocator.
- `aobf` (address ordered best fit), `aoff` (first fit), and `aowf` (worst fit): simple sequential fits with different allocation policies (best, first, and worst fit) that are able to allocate blocks according to a predefined cache set mapping. However, no useful WCET for allocation and deallocation requests can be given.
- DLMalloc: Doug Lea's allocator [10] which is considered to be the best general purpose dynamic memory allocator. However, this allocator is neither cache-aware nor does it provide useful WCET bounds for allocation requests.

We also measure the maximum amount of memory live, i.e. allocated, contemporaneously for the different benchmarks. Comparing Doug Lea's allocator and `TLSF` gives a good impression of the (isolated) costs in terms of memory consumption for constant response times. I.e., the spatial costs for switching from a *best fit* strategy to a *good fit* strategy in order to achieve constant allocation times. Our sequential fit allocators isolate the spatial costs of enforcing a certain, statically fixed cache set mapping on allocations. The difference between the maximum amount of live memory and `DLMalloc`'s memory consumption illustrates the spatial costs inherent to dynamic memory allocation; even without further demands for constant response times and cache guarantees.

For an unbiased comparison, we want to compare just the (de-) allocation routines without the actual program computations for the different allocators. Therefore, we record a trace of allocations and deallocations during one run of the benchmark application. In a subsequent step, we use this trace to synthesize a one-path program which only consists of

these allocations and deallocations. This program is then compiled once for each allocator.

Since the used subset of the MiBench benchmark suite may not be as representative as the suite as a whole, we added three additional, synthetic benchmarks. These benchmarks are artificial traces which describe different types of typical behaviours of hard real-time systems. Due to their artificial character, these traces only cover some basic characteristics of real programs. The three characteristics we used for our evaluation are suggested by [14], which also points out the weaknesses of generating synthetic, randomized (de-) allocation sequences. Namely that real programs simply do not behave randomly, but exhibit regularities that a dynamic memory allocator may exploit. Hence, results from randomly generated (de-) allocation sequences generally tend to be overly pessimistic. One typical behavioural pattern is having all allocations in a set-up phase. After this phase, the application *works* on these allocated objects without allocating more objects. This behaviour is covered in the trace called *ramp*. Another typical behaviour pattern consists of round-wise allocations. This pattern is widely found in reactive systems. These programs often run in a loop and everything which gets allocated during one iteration gets deallocated in the same iteration. This behaviour is modelled in the trace called *peak*. The third behavioural pattern that we consider is a combination of the two patterns discussed so far. I.e., there is a base of allocated objects on which the program works, but there are also additional allocations and subsequent deallocations per iteration as in the *peak* pattern. This pattern is implemented in the trace called *plateau*.

Life spans and requested block sizes are randomly selected according to an exponential distribution with rate parameter $\lambda = 0.25$. However, we shifted this distribution such that we have 1 as the smallest possible life span. The random values were furthermore multiplied by 4 to obtain reasonable, aligned block sizes. The programs *ramp* and *peak* run until 10,000 allocations are performed. The *plateau* in the third program consists of 1,500 allocations on top of which 10,000 allocations and deallocations are performed. The additional cache set arguments of `PRADA` and `CAMA` are selected according to the same heuristics used in [6]. Those heuristics are intentionally very simple, with the intention that any programmer would use a heuristics at least as good. We use two simple heuristics A and B depending on the benchmark application. Heuristics A assumes that memory is never deallocated and just put consecutively in memory. It then simulates this behaviour and sets cache set arguments to the cache set that the start addresses of allocated blocks are mapped to in its simulation. Heuristics B simply returns cache set a n-times, then n-times cache set $(a + 1)$ and so on. This heuristics assumes that n successively allocated memory blocks fit into one cache line. We used heuristics A for the `Susan` test cases as well as for all synthetical benchmarks. For `Dijkstra` and `Patricia` test cases, heuristics B was used.

The number of deferred actions in `PRADA` can be configured. We used two configurations for our benchmarks. One with splitting and merging completely disabled, i.e. allowing for 0 actions to be stored, and one with space for 32 actions, denoted `PRADA0` and `PRADA32`. `PRADA` and `CAMA` are configured for an architecture with 128 cache sets, with $i_{min} = 0$, $i_{max} = 18$, $j_{min} = 0$, and $j_{max} = 3$. `TLSF` used 24 base classes with 32 subclasses, each. Figure 4 shows the memory consumption for all allocators on the MiBench programs.

We observe a measurable impact of disabling splitting and merging on the MiBench test cases. On these real-life benchmarks, forcing the allocator to adhere to a given cache set mapping for allocated blocks (`aobf`, `aoff`, and `aowf`) also causes a measurable increase in memory consumption. While this is to be expected, this increase is surprisingly lower than the increased memory consumption due to constant response times, i.e., when compared to `TLSF`. Comparing our sequential fits with the (also) cache-set guided allocations to `PRADA` and `CAMA`,

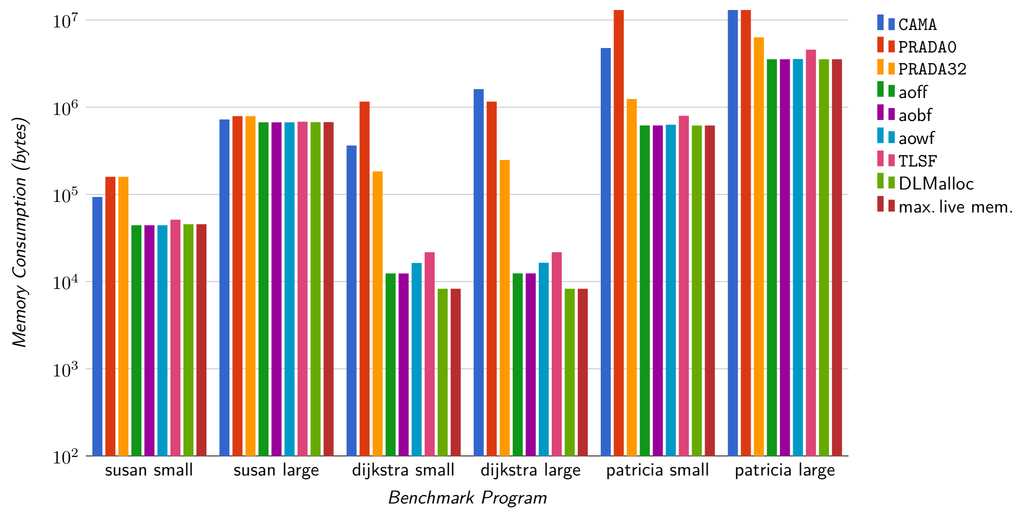


Figure 4 Memory consumption on (de-) allocation traces generated from the MiBench benchmarks.

we observe a significant jump in memory consumption. While we expect a small increase due to internal fragmentation from the employed segregated-list approach as well as the additional memory needed for `CAMA`'s descriptors, those allocators also introduce yet another kind of fragmentation. What kinds of fragmentation do exist and why do our constant-time cache-aware allocators exhibit those? General purpose dynamic memory allocators suffer from two kinds of fragmentation: *internal fragmentation* and *external fragmentation*. Internal fragmentation denotes the memory overhead when the allocator returns blocks larger than requested. This may be due to round-up block sizes, memory alignment, the allocator's inability to manage the remaining part of the block, etc. External fragmentation occurs when there is enough free memory to satisfy a request, but there is not a single block large enough. I.e. the free memory is interspersed with allocated memory. `PRADA` and `CAMA` additionally suffer from an *incomplete memory use*. This denotes the inability to find a suitable, large enough block that could be split in order to serve an allocation request simply because this block is assigned to another cache set's free lists. The term *incomplete memory use* was coined by Ogasawara to describe a similar problem of Half-Fit [12]. In Half-Fit, free blocks larger than the base size of their respective free list will not be used to serve requests for blocks of sizes larger than this base size; even if they are just one byte larger. Analytically, this leads to a worst-case memory consumption of roughly twice the maximal live memory just due to Half-Fit's incomplete memory use. While `TLSF`, `CAMA`, and `PRADA` use a similar segregated-list approach as Half-Fit, they do not *inherit* this problem. `TLSF` introduced finer grained segregated lists and always rounds up block sizes to the next segregated list base. While this simply transforms incomplete memory use into internal fragmentation, much lower analytical worst-case bounds can be given, depending on the number of second level size classes. Unfortunately, simply rounding up block sizes does not help counteracting the type of incomplete memory use occurring in `PRADA` nor `CAMA`. Still, the incomplete memory use of the cache-aware allocators can be counteracted by increasing their WCET for allocations. Currently, both allocators maintain a bit sequence indicating whether the segregated lists corresponding to the bits contain free blocks or are empty. This sequence is sorted in

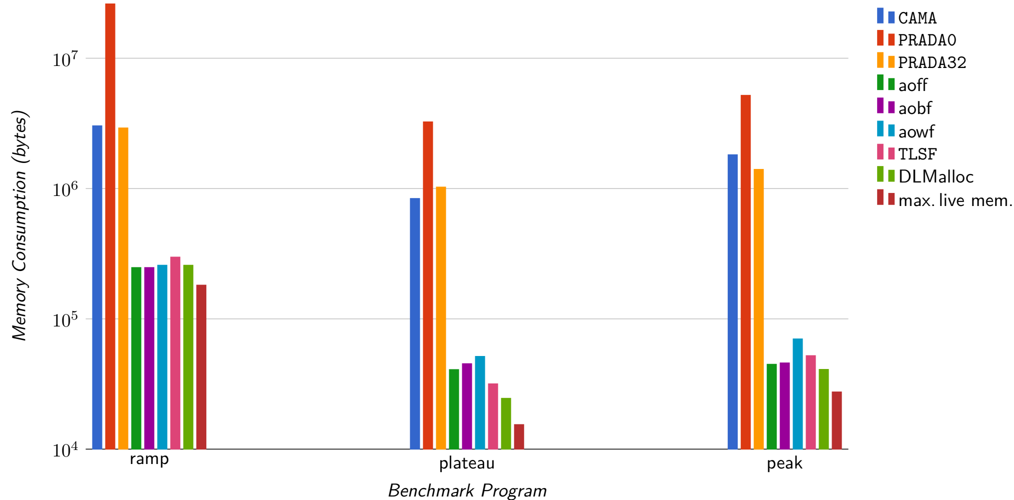


Figure 5 Memory consumption on our synthetic (de-) allocation traces.

ascending cache set numbers and (per cache set) ascending size classes. The allocator handles an allocation request for *size* bytes mapped to cache set k by computing which segregated list L would contain the smallest blocks large enough to satisfy this request. The bit sequence is then read and the first bit set to 1 is searched within the sub-string starting at the bit associated with L and ending with the bit associated with the list containing the largest free blocks whose starting addresses are mapped to cache set k. If no such bit is found, we would like to also consider other cache sets and search for larger block to split to prevent incomplete memory use. We can do this in constant time by having a second sequence of bits with the same semantics but different order. This sequence is sorted by descending size classes and (per size class) descending cache sets. On this sequence, we again search for the first bit set and take the first block from the free list corresponding to this bit and check whether it can be split to yield a block suitable to serve the original request. In other words, if the allocator's constant time *good fit* approach finds no suitable block, it reverts to a slower (in the worst-case the whole bit sequence is read), but still constant time *bad fit* approach. This fall-back mechanism is, however, not implemented yet.

Figure 5 shows the actual memory use for our randomly generated traces. On these synthetical traces, an even larger impact on the memory consumption is observable when splitting and merging is disabled. We also observe that enforcing a statically predefined cache set mapping raises memory consumption more than ensuring constant response times on these traces. Also, incomplete memory use turns out to be again the greatest source of memory waste.

4 Related Work

Dynamic memory allocators with bounded worst-case execution times have been investigated for many years. The binary buddy system is a long-known allocation algorithm whose WCET can be bounded by a constant. However, it may suffer from a relatively high internal fragmentation. The first dynamic memory allocation algorithm especially aiming at satisfying the requirements of real-time applications, `Half-Fit`, was proposed by Takeshi Ogasawara in 1995 [12]. His segregated lists approach was further refined in `TLSF` [11]. The first real-time

allocator also considering its cache effects was CAMA [8, 6].

Chilimbi et al. proposed a so-called cache-conscious memory allocator (ccmalloc), however, they aimed to improve program execution times [1]. Chilimbi's ccmalloc also takes an additional argument like CAMA and PRADA. However, instead of a fixed cache set, ccmalloc takes a pointer to an existing object that is likely to be accessed contemporaneously with the object to be allocated. ccmalloc achieves its goal by trying to allocate the newly requested storage next to the one pointed to by its second argument. As a result, newly allocated storage is often located in the same cache set as the referenced one.

Besides efforts to make memory allocators more predictable, automatically transforming dynamic memory allocation into static memory allocation was proposed as a means to allow programmers to employ dynamic memory allocation in real-time applications. Approaches to algorithmically find suitable static allocations schemes for a given program with dynamic allocation are proposed in [7] and [5].

5 Conclusions

The contributions of this paper are twofold. We propose PRADA, an alternative approach to cache-aware dynamic memory allocation. We also present a small case study investigating the sources of fragmentation and general spatial costs of dynamic memory allocation in real-time applications.

PRADA overcomes the disadvantages of general purpose dynamic memory allocators in hard real-time systems. Its implementation is simpler than that of CAMA. For the proposed allocator, a tight bound on the WCET for allocations and deallocations can be derived. The effect of allocations and deallocations on the cache state is bounded to a single cache set. This introduces predictable cache behaviour that does not hinder a static cache analysis to derive precise information about an application's cache performance. Which, again, can be used by a timing analysis to derive tight WCET bounds for the application. PRADA achieves this predictability by deferring a fixed amount of actions (splits/merges) which would cause unpredictable behaviour if always directly executed. This bound can be configured. Lower bounds may yield higher fragmentation, but lower WCET estimates. Resources on embedded hardware are restricted. And the possibility of configuration may widen the space of possible applications of our allocator.

With respect to memory consumption, we make several observations:

- Enforcing a cache set mapping on dynamically allocated objects does not necessarily significantly increase the application's memory consumption.
- CAMA's most general approach to immediately execute splits and merge (and never *drop* such an action) may not be needed in current real-time applications to keep memory consumption low. However, generally, deferred splits and merges cause higher memory usage[14], so once real-time applications become more and more complex, this may change.
- Completely disabling splitting and merging does significantly increase the memory consumption, even for (generally simpler) current real-time applications.
- Current approaches pay for strong guarantees about their cache influence with potentially drastic increases in memory consumption due to *incomplete memory use*. However, there is a potential trade-off to reduce this type of fragmentation at the price of increased WCET bounds and increased, although predictable cache usage.

References

1 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, 2000.

2 Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. Ieee, 2001.

3 Florian Haupenthal. PRADA: Predictable Allocations by Deferred Actions. Bachelor's thesis, Saarland University, 2012.

4 Reinhold Heckmann, Christian Ferdinand, Absint Angewandte, and Informatik Gmbh. Worst-Case Execution Time Prediction by Static Program Analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society, 2004.

5 Jörg Herter and Sebastian Altmeyer. Precomputing Memory Locations for Parametric Allocations. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 125–136. Austrian Computer Society, July 2010.

6 Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011.

7 Jörg Herter and Jan Reineke. Making Dynamic Memory Allocation Static To Support WCET Analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

8 Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-Aware Memory Allocation for WCET Analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

9 Daniel S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Commun. ACM*, 16(10):615–618, October 1973.

10 Doug Lea. A Memory Allocator. unix/mail, 1996.

11 Miguel Masmano, Ismael Ripoll, Jorge Real, Alfons Crespo, and Andy J. Wellings. Implementation of a Constant-Time Dynamic Storage Allocator. *Software: Practice and Experience*, 38(10):995–1026, 2008.

12 Takeshi Ogasawara. An Algorithm with Constant Execution Time for Dynamic Storage Allocation. *Real-Time Computing Systems and Applications, International Workshop on*, 0:21, 1995.

13 James L. Peterson and Theodore A. Norman. Buddy Systems. *Commun. ACM*, 20:421–431, June 1977.

14 Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Henry G. Baker, editor, *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer, 1995.

Static analysis of WCET in a satellite software subsystem*

Jorge Garrido, Juan Zamorano, and Juan A. de la Puente

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

Abstract

This paper describes the authors' experience with static analysis of both WCET and stack usage of a satellite on-board software subsystem. The work is a continuation of a previous case study that used a dynamic WCET analysis tool on an earlier version of the same software system. In particular, the AbsInt aiT tool has been evaluated by analysing both C and Ada code generated by Simulink within the UPMSat-2 project. Some aspects of the aiT tool, specifically those dealing with SPARC register windows, are compared to another static analysis tool, Bound-T. The results of the analysis are discussed, and some conclusions on the use of static WCET analysis tools on the SPARC architecture are 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, static analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2013.87

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.[1]

The software for the mission is being developed by the Real-Time Systems Group at UPM (STRAST)[2], using a model-driven approach [1] that enables auto-generated functional code from different modelling tools, including Simulink,[3] to be included as source code modules. The software structure supporting the concurrency and real-time aspects of the system is also automatically generated from high-level models using a pattern-based approach [3, 12, 11]. The resulting source code is written in Ada 2005 [9] with the Ravenscar profile restrictions [4]. Functional C code is integrated with the Ada code using the standard Ada support for C interfacing.

The hardware platform is based on a LEON computer board [7]. The executable code is generated by means of the GNATforLEON compilation chain [13], which includes the ORK real-time kernel [5].

European software standards for space systems require schedulability analysis to be carried out on on-board real-time systems [6]. The Ravenscar profile enables such analysis

* This work has been partly funded by the Spanish Ministry of Economy and Competitiveness (MINECO), project TIN2011-28567-C03-01 (HI-PARTES).

[1] `http://www.idr.upm.es/tec_espacial/upmsat2-eng/01_UPMSAT2.html`
[2] `www.dit.upm.es/str`
[3] `http://www.mathworks.com/products/simulink`.

© Jorge Garrido, Juan Zamorano, and Juan A. de la Puente;
licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 87–96

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to be statically performed using well-known response-time analysis methods [10, 2], which require the worst case execution time (WCET) of each task to be known. Since WCET analysis may be complex and the choice of the best approach is not straightforward [15], a case study was chosen in order to assess the suitability of some available tools. The worst-case execution time of one of the UPMSat2 subsystems, ADCS (Attitude Determination and Control Subsystem) was first analysed with RapiTime,[4] a dynamic analysis tool, and the results were presented in a previous paper [8].

This paper describes further work in analysing the ADCS subsystem using a static WCET analysis tool, the aiT tool by AbsInt, which is part of the a^3 analysis toolchain.[5] The results of the analysis are discussed and compared with those obtained with RapiTime.

The paper is organised as follows: Section 2 describes the UPMSat-2 platform, and the relevant architectural characteristics for the rest of the paper. Section 3 describes the SPARC register windows and the way the AbsInt tools deal with them. The approach is compared to that of Bound-T, another static analysis tool. The a^3 tools are described in some detail in section 4. Section 4.2 presents the results obtained from the static analysis, which are compared with those obtained from the dynamic analysis tool. Section 5 presents an alternative approach to modelling the SPARC register window, and its application to three different scenarios. Finally, section 6 presents the conclusions of the analysis.

2 UPMSAT-2 platform

2.1 Computer board

The engineering model used for this study is the same as in [8]. It is based on a GR-XC3S1500 Spartan3 development board[6] with a LEON2 processor[7] at 40 MHz with a 5-stage pipeline and 64 MB of SDRAM.

2.2 SPARC architecture

SPARC (Scalable Processor ARChitecture) is a reduced instruction set architecture (RISC) originally developed by Sun Microsystems [14]. The LEON family of processors is a 32-bit synthesizable VHDL processor core that implements the SPARC V8 architecture. The first LEON processors were originally developed by the European Space Agency in order to provide a stable architecture for European space projects. The aim of the architecture is to provide a platform for which compiler optimization can be easily performed, so that short time-to-market development schedules can be achieved. To this end, the architecture has been simplified with some special characteristics, such as a reduced set of instruction formats, all 32 bits wide and with only two addressing modes, and a separated and configurable floating-point register file. Other relevant aspects of the architecture include a big endian byte ordering, a vectored trap table, as well as separated coprocessor, multiprocessor and floating-point instruction sets.

One of the most distinctive features of the architecture is the use of register windows for handling local storage areas and parameter passing. This feature has a strong influence on the WCET, as discussed in the next section.

---

[4] `http://www.rapitasystems.com/rapitime`
[5] `http://www.absint.com/a3/`
[6] `http://www.pender.ch/products_xc3s.shtml`
[7] `http://www.gaisler.com/leonmain.html`

3 SPARC register windows

3.1 Overview

The SPARC architecture defines two types of registers: general-purpose registers and control/status registers. At any given time, 32 registers of 32 bits are provided for the programmer use. They are logically divided into four sets of 8 registers each: *global*, *in*, *local*, and *out* registers. Physically, they are divided into a set of 8 global registers and an implementation-dependent number of 16-register sets. Each 16-register set is in turn logically divided into 8 *in* registers and 8 *local* registers. The remaining 8 registers, the *out* registers, overlap with the *in* registers of the adjacent register set, thus making them accessible from the current set. Each set of 32-bit registers that can be identified in the processor is called a *register window* (see [14] for a detailed description).

The four register sets in a register window are used as follows:

- *Global* registers are shared by all routines.
- *In* registers are used to receive the arguments from the calling routine and return the result at the end of the current routine execution.
- *Local* registers are used to store partial results during the current routine execution.
- *Out* registers used to pass arguments to called functions and receive return values from them.

The aim of this division is to provide a higher number of registers and provide isolation for the local registers in each function. A register window mimics the top of the stack for the currently running routine, thus saving stack access time.

The number of register windows or register sets is implementation-dependent and ranges from 2 to 32. If the nesting level is deep enough, after a sequence of calls, the callee may not have a free register window to use. In that case, a register overflow trap occurs, and the run-time kernel is responsible for providing a new register window to the routine. To this end, a circular buffer of the register windows is commonly used, as shown in figure 1. In order to make space, an implementation-dependent number of register windows are saved in the stack by the overflow trap routine. After that, the call instruction can be resumed and executed successfully. In the same way, it may happen that the previous window is not available when it has to be restored after a chain of routine returns. This situation results in a register window underflow trap, which is handled by the kernel in a similar way.

The SPARC register windows mechanism is useful for reducing the average execution time of a program, but it introduces additional difficulties for computing the WCET of code sections. The reason for it is that register windows introduce a performance dependence on the execution history, in a similar way as cache memories do. The next paragraphs discuss the modelling approach that some static analysis tools currently take in order to deal with this mechanism. An enhanced approach is proposed in section 5.

3.2 Register windows in aiT

The aiT approach to modelling the SPARC register windows is overly simplistic. It assumes that the processor can create an unlimited number of register windows. Consequently, a new window can never overlap the first one, and the tool does not make use of information on window overflows or underflows. Therefore, aiT results are only correct if the code being analysed does not create more windows than available, i.e. if there are no window overflows.

On the other hand, the number of register windows created by the code can be calculated by carrying out a stack analysis with an appropriate tool, e.g. the AbsInt StackAnalizer.

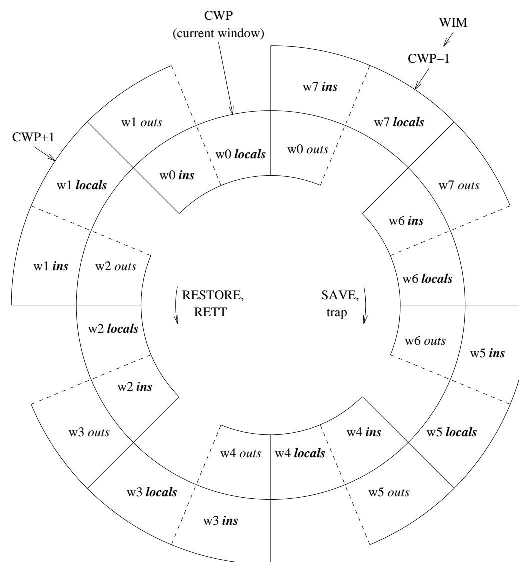


Figure 1 SPARC register window schema. In this example 8 windows are shown, but this is an implementation-dependent value that ranges between 2 and 32. Reproduced from [14].

The tool calculates a range of minimum and maximum register windows used by the code, and this information can be useful to get a better estimation of execution time. However, this is not enough to compute accurate WCET values, since more detail on the total number of windows used, and the times when they are opened or closed, is required for this purpose.

3.3 Comparison with the Bound-T approach

Bound-T[8] is another tool for static analysis of WCET. It provides an upper bound on execution time and stack usage, as well as control flow graphs and call trees, for a variety of processor architectures, including ERC32.[9] Although it does not fully supports LEON processors, it still can be used in a limited way to provide some useful information for this architecture. In particular, Bound-T uses an approach to modelling window registers that is of great interest. It is not only aware of the possibility of an overflow or underflow on each call or restore instruction, but it also offers the possibility to make assertions on the number of register windows that are in use at the entry point of each routine. This feature allows for a better analysis and understanding of the behaviour of the code with respect to register windows. A proposal on how to use Bound-T register window support to complement aiT results is described in section 5.

4 Methodological approach

4.1 Overview of the a^3 toolchain

The aiT tool is integrated in the AbsInt Advanced Analyzer (a^3) toolchain,[10] which also includes tools for stack analysis, value analysis, and other kinds of static analysis. The tools

[8] See `http://www.bound-t.com` and `http://www.tidorum.fi/`
[9] ERC32 is a predecessor of the LEON processor series, based on the SPARC V7 architecture.
[10] `http://www.absint.com/a3`

work on binary executable files, and the results are thus in principle independent of the compiler and source code language. In practice, a wide range of compilers is supported by a^3 for each target.

Static analysis tools are strongly dependent on hardware models, and thus have to be carefully configured in order to get accurate results. Apart from some processor specific features, such as pipeline behaviour, that are already included in the tool, there are other such as FPU, cache and main memory characteristics, which have to be defined by the user.

The a^3 WCET tool (aiT) generates a call graph where the nodes are labelled with their relative execution times, as shown in figure 2. In order to provide a better human interface, the labels are drawn in different colours, depending on whether they lie in the worst-case path or not. Additionally, it outputs a table with detailed information, including self and global WCET, infeasible sections of code, and different context values. An XML output is also provided in order to facilitate further processing of the results by third party tools or scripts.

4.2 ADCS case study

The controller module of the UPMSat2 Attitude Determination and Control System (ADCS) has been used as a case study for static WCET analysis with the a^3 tools. The code is the same that was previously used in an experiment on dynamic analysis carried out with RapiTime [8].

The ADCS controller code has a linear structure, and it mostly consists of arithmetic operations on vectors. In spite of its simplicity, however, the execution times of the floating-point operations is highly variable, and depends on the input values that are read by attitude sensors and processed by the control algorithm.

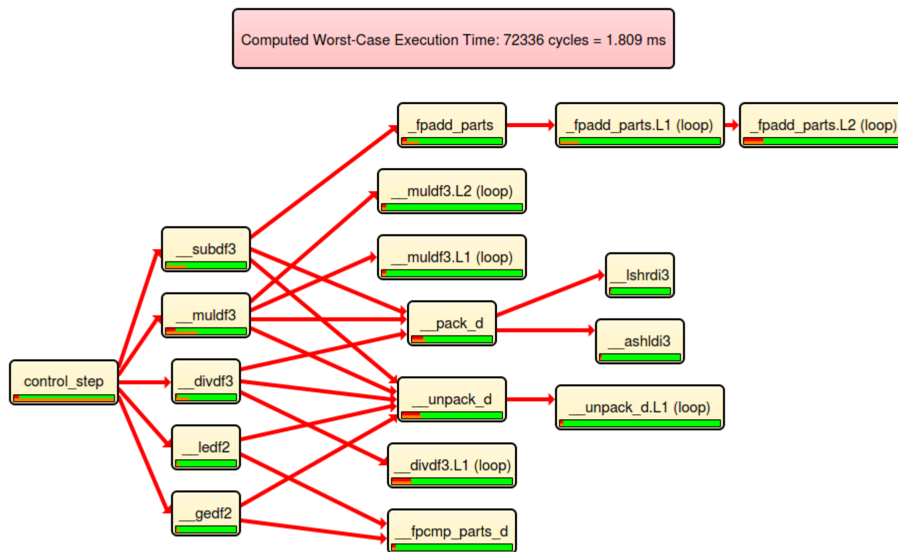


Figure 2 UPMSat-2 ADCS controller worst case execution path graph. As the controller code is linear, all the blocks are in the worst case path.

The results obtained by the a^3 tools for the controller module are shown in figure 2. The estimated WCET value is 72366 cycles, equivalent to 1.809 ms on the target platform. This result is compared with that obtained with RapiTime in table 1. It can be seen that the

Table 1 Comparison between Rapitime and a^3 results.

Tool	WCET (cycles)
Rapitime	8400
a^3	72366
a^3 with real inputs	45000

estimated WCET value is an order of magnitude higher when the static tool, a^3, is used. This could be expected to some extent, taking into account that a^3 computes its results from a model of the computer, whereas RapiTime relays on measurements. However, even considering the different approaches of both tools, the difference between the WCET values seems to be too large.

One of the main reasons for this difference are the different ways that the values of input variables are obtained. In the Rapitime experiment, values of the Earth magnetic field coming from an accurate simulation model are used (see [8] for a description of the testbench). On the contrary, in the a^3 experiment input values are automatically generated within the whole double float range. The actual range of possible values is much more restrictive, but a^3 does not provide a mechanism for imposing such restriction on this data type (although it can be done for other data types). It should be taken into account that the target LEON computer does not have an FPU, and thus the floating-point operations are implemented in software, thus making their execution time highly dependent on the operand values.

In order to compensate for this deviation from the real behaviour, the static analysis experiment has been repeated using a subset of the magnetic field input data values obtained from the simulation model. The values have been included in the model by calling the controller from a wrapper function with the simulation values declared locally. Asserting this function as an entry point, different WCET values are computed for each call. This technique provided a tighter WCET estimation, down to 45000 cycles as shown in the third row of table 1. Notice that this value has been obtained only for comparison purposes, and can not be taken as a real measure of WCET in any case. Moreover, overheads due to register window overflows and underflows have not been taken into account, which surely results in further inaccuracies. An enhanced approach including overflow and underflow overheads is developed in the next section.

5 Modelling register windows

5.1 Register windows and execution time

The fact that a^3 tools do not take into account the register windows overflows and underflows makes the WCET analysis optimistic with respect to this aspect of the SPARC architecture. In order to obtain more accurate WCET estimations, the effect of the overhead incurred by these operations on the execution time has to be appropriately modelled and accounted for in the analysis.

The first step is to get a value for the WCET of the routine that handles register window overflows and underflows, WCET_T. This value can be measured directly on the platform using a hardware monitor such as GRMON.[11] The CPU cycles gap between the first and

[11] `http://www.gaisler.com/index.php/products/debug-tools/grmon`

second call or restore instruction provides such a measurement, including memory accesses. Since the handling routines are linear, there is no cache, and further traps are disabled during their execution, we can safely assume that their execution times are constant.

The run-time kernel can deal with overflows and underflows in different ways. In particular, the number of registers that are saved and restored when a overflow or underflows occurs depends on the implementation, although it has been shown that saving and restoring one window on each overflow/underflow trap is the best approach.

Another important topic is context switches between threads. For instance, some kernels only restore the last window after a context switch. In this way, the context switch time is minimized, but in a concurrent environment each restore can potentially lead to an underflow. Moreover, each save can potentially lead to an overflow if the kernel only provides one valid window to the scheduled thread and keeps the content belonging to the preempted thread in the rest of windows. Therefore, in order to minimize the wort-case number of overflows and underflows, the full content of the register windows should be restored.

In general, the execution time of the overflow/underflow handling routine has to be multiplied by the worst-case number of overflows and underflows in the code, and the result has to be added to the WCET computed by the analysis tool in order to obtain a better estimate of the overall WCET of the code being analysed. The total overhead that can be computed in different situations is discussed in the next paragraphs. Sections 5.2 and 5.3 analyse the cases when only one window is restored or saved by the trap routines, whereas section 5.4 analyses the situation when more windows are restored or saved.

5.2 One window, no full context restore

The stack analysis tool provides the minimum and maximum number of register windows that can be used and the number of overflow/underflow traps that can occur in a single execute of a function f. Let this number be T_f, and let n_f be the number of times f is called in the worst-case path. The total worst-case number of traps occurring in f is thus

$$N_f = n_f \times T_f$$

The total number of traps in the worst-case path is

$$N = \sum_{f \in F} N_f$$

where F is the set of functions.

Finally, the total worst-case overhead is

$$\mathrm{WCOH} = N \times \mathrm{WCET}_T$$

where WCET_T is the worst-case execution time of the trap handler routines.

Applying this technique to the case study, Bound-T reports a total of six functions causing overflows, with a total number of overflow traps $N_O = 25$ and a similar number of underflow traps, $N_U = 25$. The WCET is not the same for overflow and underflow traps, and the respective values are $\mathrm{WCET}_O = 156$ and $\mathrm{WCET}_U = 188$ cycles. Hence the total overhead caused by the overflow and underflow traps is

$$\mathrm{WCOH} = 25 \times 156 + 25 \times 188 = 8600\,\text{cycles}$$

This value is significant, as it amounts to an 11.56 % of the WCET computed by a^3 and a 102.38 % of that computed by Rapitime.

5.3 One window, full context restore

The ORK kernel used in the UPMSat-2 software saves and restores only one window per trap, as previously considered, but in this case the full state of the registers is restored on each context switch. Therefore, the worst-case number of overflows in a code section equals the maximum depth of register windows it can create. This number is usually much lower than that obtained in the previous situation. Underflows can only occur if the depth of windows saved in the stack is greater than the number of physical windows.

For the ADCS case study, both a^3 and Bound-T reported a maximum register window depth $D_W = 3$. This means that the maximum number of overflows is $N_O = 3$. Since the LEON processor has 8 real register windows, no underflow can occur. The overhead is now

$$\text{WCOH} = 3 \times 156 + 0 \times 188 = 468\,\text{cycles}$$

This value is much lower than the previous one. It amounts only to a 0.63 % of the a^3 WCET value and a 5.57 % of the Rapitime value.

5.4 More than one window

The SPARC architecture enables from 2 to 32 register windows, and implementations can restore between 1 and the total number of sets. This greatly increases the complexity of the analysis, as the number of possible behaviour may become very high. A full study of all the implementation possibilities is out of the scope of this paper, but previous versions of the ORK kernel are discussed as an example.

- In ORK 1.0 only the last window was restored after a context switch. This case is the “one window with no restore” case, as any save and restore may cause a trap.
- In later ORK versions, the full register set is left as it was before the context switch. In order to achieve this result, the trap handling routines for register window overflow and underflow save or restore the full set of windows. For example, on a processor with 8 register windows, there are 7 windows available to user software. Therefore, after 7 consecutive saves a window overflow occurs. In the worst case, user code may save and restore windows in the edge of the window invalid mask,[12] generating a trap on each save or restore.

In the case study the worst case happens when the user function is called using the last valid window, and thus every procedure call causes a window overflow. This includes all calls to floating-point arithmetic routines, which are frequent. Similarly, every return from the routines causes an underflow. Therefore the overhead is just one overflow and underflow less than the “one window, no restore” case. i.e.

$$\text{WCOH} = 24 \times 156 + 24 \times 188 = 8256\,\text{cycles}$$

Since more than one window is saved or restored on each overflow or underflow trap, the execution time of the trap handling routines may increase. Therefore, the above values should be taken as a lower bound.

In all cases, the computed overhead must be added to the WCET computed with the the basic analysis of section 4.2. Table 2 summarizes the results compared to the basic analysis.

[12] The window invalid mask is a bit map of valid windows [14].

Table 2 Rapitime and a[3] results with register windows overhead (times in cycles).

Tool	Basic analysis	1 w. full restore	1 w. no restore	7 w. full restore
Rapitime	8400	8868 (+5.57%)	17000 (+102.38%)	16656 (+98.28%)
a^3	72366	72834 (+0.63%)	80966 (+11.56%)	80622 (+11.28%)

6 Conclusions

Modelling modern processors is a complex task, due to the wide range of hardware acceleration features they include, such as cache memories, memory management units, coprocessors, or multicore architectures. Although the SPARC v8 is far from new, modelling its set of register windows is not trivial and indeed complicates the task of estimating execution times on this kind of processors. In general, it can be said that the SPARC registers architecture improves efficiency and reduces the overall execution time of the application code. However, if not properly analysed, it can lead to imprecise, unsafe results.

Some software elements can also have an influence on the execution time of code running on register window architecture. In particular, run-time kernels may deal with window overflows and underflows in different ways. Such differences may result in different levels of overheads on the WCET values, as shown in section 5. Other sources of inaccuracy may arise that make it difficult for analysis tools to predict the execution behaviour of code. For example, loop bounds may depend on input values that are unknown at the time of analysis. A way to cope with this issue is to use assertions to restrict the number of possible execution paths. However, assertions are often hard to establish and prone to errors, and thus unsafe.

The scenarios analysed in section 5 and summarized in table 2 show that different hardware and software implementations of the SPARC register windows produce different levels of overhead in the case execution time that must be taken into account for the estimation of WCET.

Acknowledgements

The authors would like to thank AbsInt and Tidorum for their active collaboration and the support provided. We would especially like to express our gratitude to Christian Hümbert from AbsInt, Enrico Mezzetti from Università degli Studi di Padova, and Niklas Holsti from Tidorum, for their support and personal implication.

References

1 Alejandro Alonso, Emilio Salazar, and Juan A. de la Puente. Design of on-board software for an experimental satellite. In *Jornadas de Tiempo Real — JTR-2013*, 2103.

2 Neil Audsley, Alan Burns, Rob Davis, Ken Tindell, and Andy J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8(3):173–198, 1995.

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

7 Gaisler Research. *LEON3 – High-performance SPARC V8 32-bit Processor. GRLIB IP Core User's Manual*, 2012.

8 Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012.

9 ISO. *Ada Reference Manual ISO/IEC 8652:1995(E)/TC1(2000)/AMD1(2007)*, 2007. Available at `http://www.adaic.com/standards/ada05.html`.

10 Mathai Joseph and Paritosh K. Pandya. Finding response times in real-time systems. *BCS Computer Journal*, 29(5):390–395, 1986.

11 Enrico Mezzetti, Marco M.Panunzio, and TullioVardanega. Preservation of timing properties with the Ada Ravenscar profile. In Jorge Real and Tullio Vardanega, editors, *Reliable Software Technologies – Ada-Europe 2010*, number 6106 in LNCS, pages 153–166. Springer-Verlag, 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 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.

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

15 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.

Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources

Leonidas Kosmidis[1,2], Tullio Vardanega[3], Jaume Abella[2], Eduardo Quiñones[2], and Francisco J. Cazorla[2,4]

1 **Universitat Politècnica de Catalunya**
2 **Barcelona Supercomputing Center**
3 **University of Padova**
4 **Spanish National Research Council (IIIA-CSIC)**

Abstract

The use of complex hardware makes it difficult for current timing analysis techniques to compute trustworthy and tight worst-case execution time (WCET) bounds. Those techniques require detailed knowledge of the internal operation and state of the platform, at both the software and hardware level. Obtaining that information for modern hardware platforms is increasingly difficult.

Measurement-Based Probabilistic Timing Analysis (MBPTA) reduces the cost of acquiring the knowledge needed for computing trustworthy and tight WCET bounds. MBPTA based on Extreme Value Theory requires the execution time of processor instructions to be independent and identically distributed (i.i.d.), which can be achieved with some hardware support. Previous proposals show how those properties can be achieved for caches. This paper considers, for the first time, the implications on MBPTA of using buffer resources. Buffers in general, and *first-come first-served (FCFS) buffers* in particular, are of paramount importance as the complexity of hardware increases, since they allow managing contention in those resources where multiple requests may be pending. We show how buffers can be used in the context of MBPTA and provide illustrative examples.

1998 ACM Subject Classification D.2.4 Software Engineering: Software/Program Verification

Keywords and phrases WCET, Buffer, Probabilistic Timing Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2013.97

1 Introduction

There is an increasing need for high guaranteed performance in Critical Real-Time Embedded Systems (CRTES) industry such as automotive, space, and aerospace. To respond to this demand, more complex hardware is used, which allows increasing performance per chip unit, which in turn enables running more functionalities per chip, thus reducing size, weight and power consumption costs at system level.

Probabilistic Timing Analysis (PTA) [4][3] has recently emerged as an alternative to conventional static (STA) and measurement-based timing analysis (MBTA) techniques [11]. Although PTA is not as mature as STA and MBTA yet, it promises to reduce dependence on execution history. This is done by randomising the timing behaviour of some processor resources, which reduces the amount of information needed to obtain tight WCET bounds in comparison to other timing analysis approaches.

PTA provides WCET estimates with an associated probability of exceedance (pWCET). In analogy to the practice that expresses reliability for embedded safety-critical systems in terms

© Leonidas Kosmidis, Tullio Vardanega, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla; licensed under Creative Commons License CC-BY
13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 97–108

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of allowable probabilities of hardware failures, PTA extends this notion to timing correctness by determining the probability with which a given WCET bound can be exceeded during system operation. PTA aims to obtain pWCET estimates for arbitrarily low probabilities, so that even if the chosen pWCET estimate can be exceeded, it would be with low enough probability (e.g., in the region of 10^{-12} per hour of operation, largely below the required probability of hardware failures). PTA can be applied either in a static (SPTA) [3] or measurement-based (MBPTA) [4] manner. This paper focuses on the latter, which is more easily amenable to industrial practice.

Contribution. PTA can be applied to hardware/software platforms where the ETP per instruction can be derived. PTA-compliance has been achieved so far for processors equipped with cache memories [5, 6]. In this paper we extend this to buffer resources. Buffers allow managing contention in those resources where multiple requests may be pending, decoupling the speed at which requests are sent and processed. Our contribution is threefold: (1) We prove that buffers can be used while preserving compliance with MBPTA requirements. Unlike other resources like caches that need to be time-randomised in order to work properly with MBPTA, buffers require no changes to be used with MBPTA. (2) We provide a new classification of hardware resources and describe how they can be adopted with MBPTA. (3) We show that, although buffers and any other complex resource in general can create dependences across instructions, they can be analysed by MBPTA as long as those dependences, regardless of their nature, whether deterministic or probabilistic, stay the same at analysis and during operation. For buffers in particular, we show how the dependences they create across instructions are purely probabilistic in a MBPTA-compliant processor and do not change between analysis and deployment.

2 Background

Unlike previous analysis techniques that provide a single WCET value per program, PTA provides a distribution function that upper bounds the execution time of the program under analysis, guaranteeing its execution time only exceeds the corresponding execution time bound with a probability lower than a given target threshold (e.g., 10^{-16} per activation). In this way the pWCET is defined as the execution time bound with its associated exceedance probability.

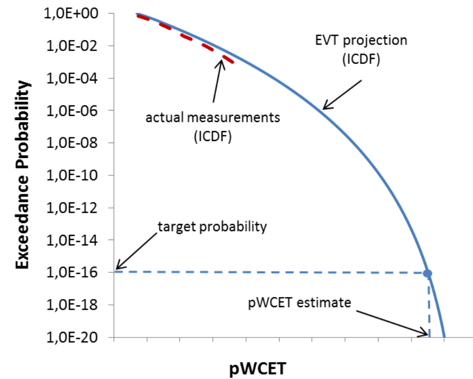


Figure 1 Example of the pWCET curve.

The timing behaviour of a program (and equivalently that of individual processor instructions) is represented with an Execution Time Profile (ETP). An ETP is the probability distribution function describing the different execution times that the program can take (the latencies, for processor instructions) and their associated probabilities. That is, the timing behaviour of a unit of execution (program, instruction) can be defined by the pair of vectors $(\vec{l}, \vec{p}) = \{l_1, l_2, ..., l_k\}\{p_1, p_2, ..., p_k\}$, where p_i is the probability the program/instruction having latency l_i with $\sum_{i=1}^{k} p_i = 1$.

The ETP for a program (resp. instruction) may vary with the program input sets that lead to different execution paths. Furthermore, the ETP for an instruction may vary

across multiple uses as execution events (e.g. previous accesses to memory) affect the state-dependent timing behaviour of that instruction. In Annex I we analyse those aspects showing that (1) the effect of past random events affect the ETP of an instruction in a probabilistic manner, whereby PTA continue to be applicable; and (2) each PTA technique has its own mechanisms to address the multiple execution path problem.

MBPTA requires the hardware to guarantee that each operation (at the granularity of processor instructions or below) has its own ETP. However, unlike SPTA, which needs all ETPs to be known, MBPTA only requires those ETPs to exist. In other words, if execution times were collected by rolling a die, SPTA would need to know the number of faces of that die, the value on those faces, and their individual probabilities of occurrence. Conversely, MBPTA would derive pWCET estimates by simply rolling the die, that is to say, by executing the program a given number of times, observing the resulting execution times and treating them with *Extreme Value Theory* (EVT) [4, 8] to a trustworthy and tight upper bound to the tail of the observed execution time distribution. By doing so, MBPTA provides *pWCET* estimates for arbitrarily low *target probabilities*. Figure 1 shows a hypothetical result of applying EVT to a collection of 1,000 observed execution times. The dotted line represents the inverse cumulative distribution function (ICDF) derived from the observed execution times. The continuous line represents the projection obtained with EVT.

2.1 Probabilistic Behaviour of Simple Processor Resources

Processor resources can be regarded as abstract components that process requests. Each such request has a distinct service time or latency, which can either be fixed or variable.

Jitter-centric resource classification. We term *jitter* the difference between the best and worst possible latency of any resource. Resources can be then classified depending upon whether they exhibit jitter or not. *Jitterless resources* have a fixed latency, independent of the input request or of the past history of service of the resource. Many hardware resources in current processor architectures can be classified as jitterless. Other resources, for instance cache memories, have a variable latency and hence are *jittery resources*; their latency depends on their history of service, i.e., the execution history of the program, the input request, or a combination of both. Jittery resources have an intrinsically variable impact on the WCET estimate for a given program. The significance of this impact depends on the magnitude of the jitter, the program under study, and the analysis method. A way to deal with jittery resources in the absence of timing anomalies is to assume that all requests to those resources *incur the worst-case latency* [9]. This is acceptable if the cumulative impact on the WCET from assuming the worst-case jitter for the resource is deemed low enough by the system designer. If taking the worst latency is not acceptable, then the timing behaviour of the resource must be randomised. This is the case of the cache, since taking its worst latency would greatly amplify the pWCET estimate. Several works propose time-randomising caches to reach both, probabilistically analysable behaviour and high guaranteed and average performance [5, 6].

ETP and jitter. Jitterless resources are easy to model for all types of static timing analysis. Building the ETP of a simple instruction that uses a single resource, requires knowing only whether the resource in question is jitterless (information implicit in the instruction) or whether the instruction is part of a sequence of instructions that must incur a delay when using a jitterless resource (information implicit in the architecture). With proper path and pipeline analysis, the types of the resources can be determined. Of course, measurements obtained from program runs that only use jitterless resources will perfectly capture their

Table 1 Code example with hit/miss probabilities for the instruction and data caches.

instruction id	instruction type	IL1		DL1	
		hit prob.	miss prob.	hit prob.	miss prob.
$i1$	LD	1.0	0.0	0.9	0.1
$i2$	ADD	0.7	0.3	-	-
$i3$	ADD	0.6	0.4	-	-
$i4$	ADD	1.0	0.0	-	-

constant impact on execution time. If the instruction accesses a jittery resource whose worst-case latency is acceptable for the designer, forcing that resource to always take the longest latency would be a simple yet effictive way to make the resource PTA-conformant: the ETP of that resource would have a single latency value (its worst case) with probability 1, i.e. 100% probability of maximum latency, leading to a upper-bounded deterministic jitter.

Instructions may access multiple resources during their execution, and those resources can be arranged in different manners, e.g. sequentially or in parallel. Under each arrangement, the ETP of those resources can be properly combined to derive the ETP of the instruction. To that end, several forms of convolution, $\otimes$ [3], can be used either adding latencies (sequential arrangements) or picking the maximum latency of the elements convolved (parallel arrangements).

3 Complex Processor Resources

However, the taxonomy presented in previous section does not cover *buffers*, which in fact are in widespread use in modern processor architectures. Buffers are used to temporarily hold some information decoupling the timing of the sender and the receiving elements. If a buffer is full it may create stalls that propagate backwards in the pipeline of the processor, thus potentially increasing the execution time and affecting WCET.

3.1 Timing Behaviour of a Buffer in a Time-Randomised Architecture

For the sake of illustration, let us assume an architecture with two stages (fetch and execute) that respectively access instruction and data caches (IL1 and DL1 for short). Caches deploy random placement and random replacement [5], which enable computing a probability of hit/miss for every access. In between both stages there is a 2-entry buffer (see Figure 2). In case of hit in both caches and if the buffer is available, an instruction takes 3 cycles: Fetch (F), buffer (b) and Execute (E).

Further assume that we execute the program with four instructions shown in Table 1, whose hit and miss probabilities for each cache are shown next to each instruction. For this example, $i1$ always hits in IL1 and has a 0.9 hit probability in DL1. The remaining instructions do not access DL1.

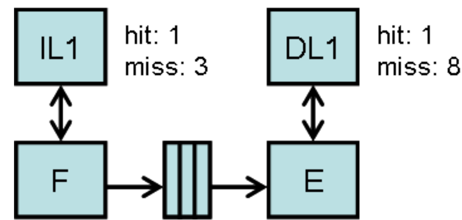


Figure 2 Processor setup considered in Section 3.1.

In the program fragment shown in Table 1, $i1$ may introduce some delay in the execution of the program when accessing DL1. In particular, if it misses in the data cache it will cause a longer delay than if it hits. Note that the IL1 hit probability of $i1$ is 100%, hence always hitting in IL1. $i2$ and $i3$ may introduce some delay when accessing IL1 only since they are not memory operations.

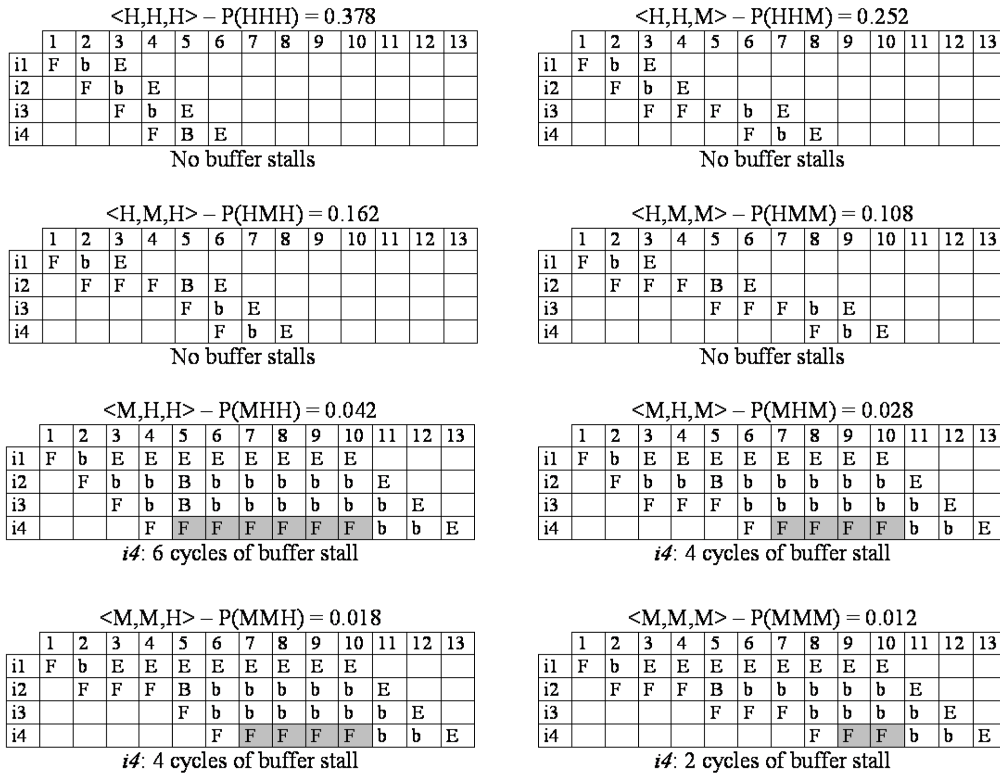


Figure 3 Potential chronograms based on the outcome of the different cache accesses. (<DL1-i1 IL1-i2 IL1-i3>) Grey rectangles show the cycles in which the processor is stalled due to the buffer.

In Figure 3 we depict the 8 different chronograms for each one of the combinations of hits and misses in IL1 and DL1 of all 4 instructions. The x-axis shows the cycles of execution while the y-axis shows each instruction. Each rectangle represents the stage in which each instruction is in each cycle: 'F' fetch, 'b' buffer and 'E' execute. We use the vector *<DL1-i1, IL1-i2, IL1-i3>* to describe the outcome of each DL1 and IL1 access, being H a hit and M a miss. For instance <HHH> is the event 'i1 hits in DL1' and both 'i2 and i3 hit in IL1'. Similarly P(HHH) is the probability of that event to happen. Note that $i1$ and $i4$ have IL1 hit probability of 100% so for this reason IL1i1 and IL1i4 do not appear in the vector.

The key appreciation we do in the behaviour of the buffer is the following: given a set of fixed initial conditions (e.g. empty state of the pipeline) each different combination of probabilistic events (e.g. DL1 and IL1 accesses) leads to exactly one fully-deterministic behaviour of the buffer. If we compare different outcomes of probabilistic events, we observe that the buffer introduces a different number of stall cycles (0, 2, 4 or 6 cycles) for each combination of probabilistic events. The number of stalls and the particular cycles in which the stalls occur may repeat in different sequences of outcomes of the probabilistic events occurring (for instance cases <M,H,M> and <M,M,H>). However, for a particular sequence of random events the behaviour of the buffer is fully deterministic: all data dependences, which are given by the sequence of instructions that are executed and their order. Given that MBPTA works or a per-path basis, in each path the sequence of instructions executed is known and fixed across runs of the same path.

The initial conditions can be caused to a fixed state by flushing the state of the resource prior to its use. Alternatively, it might be possible to probabilistically determine the state

left by previously executing code. We refer the reader to [7]. for more details.

In order to better understand this phenomenon, Figure 4 depicts, for the same example shown before, the *probability tree* for the states of the processor in each cycle. In cycle 1 *i1* is fetched. In cycle 2 *i1* is stored in the buffer while *i2* is fetched. Accessing DL1 is a random event that has two outcomes hit/and miss, and hence spawns into two possible probabilistic states, which generates a new branch in the probability tree as shown in cycle 2.

In the left branch, during cycle 3, *i1* accesses DL1 while *i3* accesses IL1. Both are probabilistic events that generate 4 new branches in the probability tree. Similarly, in the right branch in cycle 3, *i1* accesses DL1 generating two branches in the probability tree.

As shown, the variability in the execution time increases the number of potential probabilistic states that we can reach. It is interesting noting that all the execution time variability can only be introduced by probabilistic events.

In this diagram, the stalls due to the buffer are shown with grey boxes. Unlike caches that introduce probabilistic variability, and hence generate new branches in the probability tree, buffer stalls cannot produce probabilistic variability, instead *buffer variability has no effect on the probability of each execution time to occur*. Therefore buffers *cannot create probabilistic jitter* but simply propagate jitter or, in other words, given a sequence of outcomes for all probabilistic events the delay of the buffer resources is fully deterministic.

Under MBPTA, the fact that buffer resources can affect the duration of the program under each combination of probabilistic events but cannot affect the probability of each combination, simplifies their analysis. As long as the execution time observations obtained sufficiently cover, in probabilistic terms, the outcome of random events, it is also enough to safely cover the effect of buffers.

3.2 Classification of Sources of Jitter

So far we regarded jitter as deterministic or probabilistic (the latter for time-randomised resources). Yet, as shown above, the jitter caused by buffers does not fit into either category; instead, it simply propagates the inbound jitter regardless of its nature.

With this insight, we classify the potential sources of jitter into 6 groups depending on the combination of two factors: (i) whether the jitter is produced solely by the event under consideration (no history dependence) or by the combination of previous events and the current one (history dependence); and (ii) whether the jitter is deterministic, probabilistic or simply propagated regardless of its source. We omit two groups for which we did not find any existing resource to fit in.

This new classification of hardware resources will help analysing whether a given resource or processor architecture is MBPTA-compliant. To that end, for each group we identify how resources in that group can be used in the context of MBPTA.

- *No history dependence + deterministic jitter.* This could be the case of a resource whose latency does not depend on the sequence of requests it has received, but on the data of each request. For instance, the floating-point unit in some processors is affected by the particular operands (data) being operated. For this type of resources we typically enforce the unit to experience always its maximum latency as explained before, which can be done deploying a simple hardware mechanism called the worst-case mode [9].
- *History dependence + probabilistic jitter.* This is the case of a time randomised cache [5]. The sequence of events between two consecutive accesses to the same data together with the initial cache state, determine the hit/miss probability of that access. Time randomised caches have been shown to be analysable with MBPTA [5].

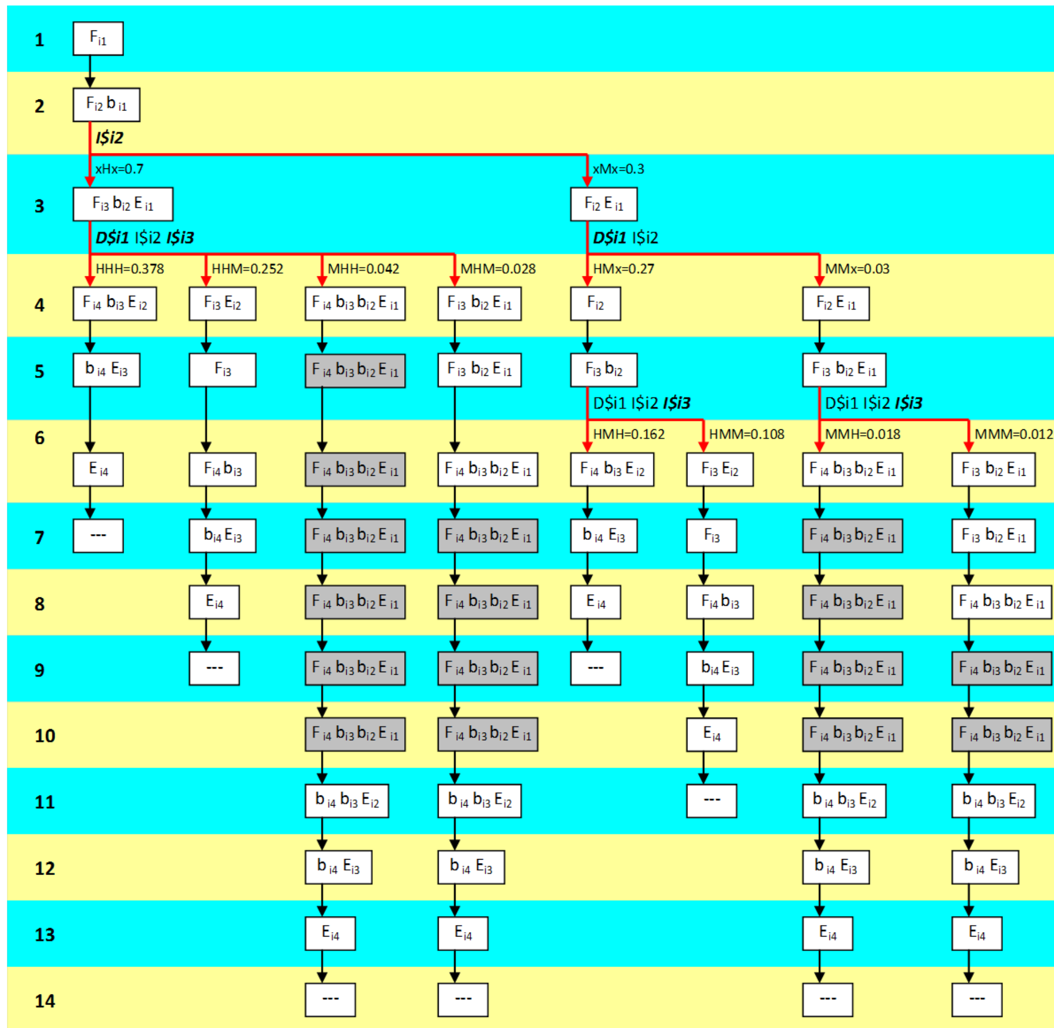


Figure 4 Processor Stage Graph.

- *History dependence + deterministic jitter.* This is the case of a deterministic cache implementing modulo placement and LRU replacement. Events may experience different latencies depending on previous history: for a given initial state and a sequence of events their latency is always the same. This type of resources is not analysable by MBPTA in general unless the factors that influence the jitter are fully under control, so that it can be known whether the observations taken to feed MBPTA cover the worst behaviour of those factors of influence. In general, the only easy way to enable the use of this type of resources in the context of MBPTA is using the worst-case mode.

- *History dependence + jitter propagation.* This is the case of a hardware buffer. A particular instruction may spend a different number of cycles in a buffer depending on previous events. However, as explained before, buffers do not create new jitter by themselves. Instead, they only propagate deterministically the effect of the jitter induced by other resources. If such jitter is probabilistic, then the stalls induced by buffers occur also with a given probability and so they are analysable with MBPTA.

3.3 Empirical Verification

Although we have described how buffers meet the MBPTA requirements if they are already fulfilled by the processor in use without buffers, in this section we verify empirically that this claim holds by testing that execution times in such a processor are independent and identically distributed, as required by MBPTA. To that end we apply the experimental methodology shown in [4].

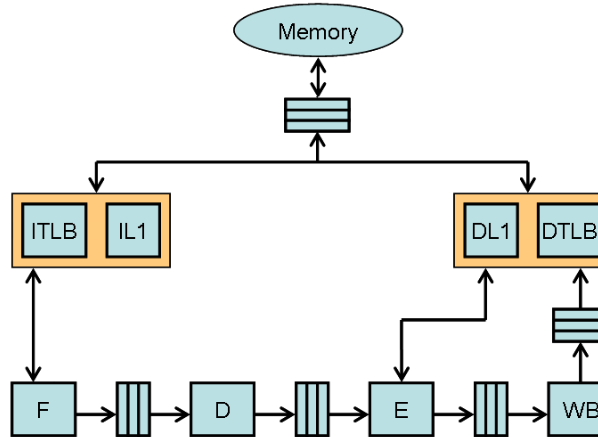


Figure 5 Processor setup considered in Section 3.3.

We consider a pipelined processor with in-order fetch, dispatch and retirement of instructions (see Figure 5). Fetch and execution stages are equipped with first level instruction and data cache memories respectively (IL1 and DL1 caches for short). Instruction and data translation look-aside buffers (ITLB and DTLB) are also in place. Buffers across pipeline stages are deployed to mitigate stalls. Similarly, a store buffer is provided to allow store instructions to retire quickly without stalling the pipeline[1]. Both IL1 and DL1 size are 4KB 8-way 16-byte line caches. Both caches implement random placement and replacement policies [5]. DTLB and ITLB are 16-way fully associative, and page size is 1KB. The latency of the fetch stage depends on whether the access hits or misses in the IL1 and ITLB: only if the access hits in both its latency is 1-cycle, and 100 cycles otherwise. After the decode stage, memory operations access the DL1 and DTLB and their behaviour is analogous to that of IL1 and ITLB. The remaining operations have a fixed execution latency (e.g. integer additions take 1 cycle).

For our experiments we use the EEMBC Autobench benchmark suite [10] that reflects the current real-world demand of automotive systems. The fact that, unlike EEMBC, real-world programs normally have multiple paths does not invalidate the conclusions of our analysis: this is so because MBPTA considers individual paths.

In order to test independence we use the Wald-Wolfowitz independence test [2]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained after running this test are below 1.96 if there is independence, otherwise are higher. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [1] as described in [4]. For a 5% significance level, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution, otherwise non-identical distribution.

Table 2 shows the results of both tests for all EEMBC benchmarks, when running each benchmark as many times as needed by MBPTA (up to 1,000 times per benchmark in our evaluation). As shown, both tests are passed in all cases.

[1] A store buffer is a particular incarnation of buffer resources.

Table 2 Independence and identical distribution tests results.

Benchmark	a2time	aifftr	aifirf	aiifft	cacheb	canrdr
Indep. test	0.90	0.10	0.27	0.11	0.51	0.21
Ident. distr. test	0.64	0.93	0.84	0.70	0.40	0.39
Benchmark	**iirflt**	**puwmod**	**rspeed**	**tblook**	**ttsprk**	
Indep. test	0.11	0.37	0.33	0.47	0.63	
Ident. distr. test	0.80	0.89	0.27	0.93	0.73	

4 Conclusions

In this paper we show that buffer resources do not create any jitter on their own but, instead, they simply propagate inbound jitter regardless of the nature of it. With this, we prove that buffers do not break PTA requirements, hence can be used in PTA-conforming processors with no change. We also provide a comprehensive classification of hardware resources and how they can be considered in the context of PTA.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROARTIS Project (www.proartis-project.eu), grant agreement no 249100. This work was partially supported by EU COST Action IC1202: Timing Analysis On Code-Level (TACLe). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the Juan de la Cierva grant JCI2009-05455.

A Annex I. Meeting MBPTA requirements

Next, we show how the existence of an ETP for each instruction in a program is a necessary and sufficient condition to make a program and a target platform analysable with MBPTA. To that end and without loss of generality we assume a processor architecture in which core operations (e.g., MUL and ADD) take a fixed latency and memory operations (e.g., LD and ST) access a fully-associative random-replacement cache [5].

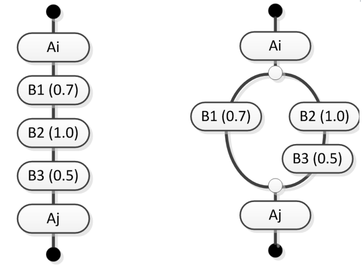
Let us assume a fully-associative cache with W ways and random replacement[2]. An approximation to the probability of hit of a given access Aj in the sequence $< Ai\ B1\ B2, ..., \ Bk\ Aj >$, where Ai and Aj access the same cache line and all Bl access other cache lines, is given by [5]:

$$P_{hit_{A_j}}(S) = \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}} \quad (1)$$

In the equation, $\frac{W-1}{W}$ is the probability of one access to evict Aj, while the exponent gives a measure of the number of evictions Aj can suffer depending on the probability of each $\{B_l\}_{l\in(1..k)}$ to miss in cache. We observe that Aj depends on execution history, i.e., $\{B_l\}_{l\in(1..k)}$. In particular, the probability of miss of $\{B_l\}_{l\in(1..k)}$ affects the probability of hit/miss of Aj. In a given run, the fact that a given Bl hits/misses in cache affects the probability of hit of Aj in that run. For instance the probability of hit of Aj in a run in which $B1$ misses is different from another run in which $B1$ hits. Hence, under each history of outcomes for $\{B_l\}_{l\in(1..k)}$, Aj may have a different hit probability.

We focus on two scenarios as depicted in Figure 6. In the first one, Figure 6(a), the whole sequence of accesses is in the same basic block, while in the second one, Figure 6(b), the sequence of accesses is spread across several basic blocks.

1) Single path. When all accesses in a sequence affecting a given access Aj are in the same basic block, they affect Aj in each run systematically, since all $\{B_l\}_{l\in(1..k)}$ are present in each run. Under each history of outcomes Aj may have a different probability of hit, and hence a different ETP.



(a) single basic block (bb) (b) branch structure (several bb)

Figure 6 Cache access sequences and distribution over different basic blocks.

Interestingly, hit/misses affecting Aj's probability of hit are random events by construction for a cache using random replacement and random placement. This introduces a probabilistic variation in the probabilities of each execution time of Aj. Hence, if enough runs are made the observed frequencies of hit/miss for each Bi and Aj will converge to their actual hit/miss probability. As a consequence, Aj can be regarded as having an ETP, where the probability of hit of Aj is that resulting from executing the program an infinite number of times. In the example in Figure 6(a), for a cache with W=8 ways the ETP of A_j is as follows: $\{l_h, l_m\}\{0.745, 0.255\}$.

Note that, if the variability that $\{B_l\}_{l\in(1..k)}$ causes on A_j is not probabilistic, which happens for instance if cache is not randomised (e.g., if modulo placement is used), the hit

[2] A similar analysis can be done for set-associative caches [5].

Table 3 ETPs of Aj under each history of outcomes.

B1-3 outcome hist.	No. of Evicts.	Prob. of that outcome	ETP of (Aj) under that outcome history
000	3	0.35	$\{l_h, l_m\}\{0.670, 0.330\}$
001	2	0.35	$\{l_h, l_m\}\{0.766, 0.234\}$
010	2	0	-
011	1	0	-
100	2	0.15	$\{l_h, l_m\}\{0.766, 0.234\}$
101	1	0.15	$\{l_h, l_m\}\{0.875, 0.125\}$
110	1	0	-
111	0	0	-

event for $\{B_l\}_{l\in(1..k)}$ is not random, so we could not derive an ETP for Aj disallowing the use of MBPTA.

Results. In the example in Figure 6(a) there is a dependence between Aj and the history of outcomes of $B1, B2$ and $B3$. In particular, for a given run the number of misses incurred by $B1$-$B3$ determines the number of random evictions carried out between Ai and Aj. The second column in Table 3 shows the number of evictions carried out under each outcome history for $B1$-$B3$. The third column shows the probability of that outcome based on the probability of miss of $B1$-$B3$. Finally, the fourth column shows the ETP of Aj assuming a fully-associative cache of 8 ways. With enough runs, the final miss probability for Aj can be computed as the addition of the probability of each possible history of outcome of $B1$-$B3$ times the probability of Aj to miss under that outcome:

$$P^m_{Aj} = \sum_{k=1}^{No.Outcomes} Prob_{Outcome_k} \times P^m_{Aj_{Outcome_k}} \tag{2}$$

that in our example results in: $P^m_{Aj} = (0.330 \times 0.35) + (0.234 \times 0.35) + (0.234 \times 0.15) + (0.125 \times 0.15)$=0.251, that accurately matches the value computed with Equation 6[3], where $P^h_{Aj} = (7/8)^{(0,7+1+0,5)}$. Hence the ETP for Aj is: $\{l_h, l_m\}\{0.749, 0.251\}$.

Therefore, although the probability of each latency of an instruction depends on its execution history – the set of outcomes of previous accesses in our case – the fact that factors of influence on its execution history are random, and hence they occur with a given probability, makes it possible to derive an ETP for the instruction. If enough samples are taken from the timing behaviour of Aj during analysis time, the observed behaviour is representative of its behaviour during deploy time. This is so because the factors of influence on Aj execution time have a random nature, so for a higher number of runs the observed frequencies of each event converge to the actual probability of the event.

2) Multiple paths. In the situation depicted in Figure 6(b) we observe that the hit probability of Aj depends on the particular path followed. Hence, the ETP of Aj is affected by: the path followed and the history of hit/misses. If Aj is reached through the left path, hence under

[3] Note that minor discrepancies are expected given that hit/miss events are not independent among them, so the hit probability computed in Equation 1 is an approximation. In fact, there are only two ways to derive the actual hit/miss probabilities: (i) Performing an infinite number of runs and measure actual probabilities, or (ii) Computing the probability of each particular cache state left by the sequence of hits and misses for previous accesses, and accumulating the probabilities for those cache states where the current cache access would result in a hit/miss.

the sequence $< Ai\ B1\ Aj >$ it has higher probability of hit than if it is reached through the right path under the sequence $< Ai\ B2\ B3\ Aj >$: $ETP_{left} = \{l_h, l_m\}\{0.911, 0.089\}$ and $ETP_{right} = \{l_h, l_m\}\{0.818, 0.182\}$. Differently to the single-path case, now there is one ETP per path leading to Aj. MBPTA provides pWCET estimates for the set of paths exercised with the input data used during the testing phase. It is also the case that MBPTA is insensitive to the frequency each path is exercised as long as each path is exercised a minimum number of times [4]. Overall, having for each instruction and path-leading-to-that-instruction one ETP preserves the i.i.d. property in the execution time of each path. MBPTA [4] samples the execution time observations obtained from each path to obtain an i.i.d. sample that covers the execution time observed for all paths.

Results. In [4] it is shown how MBPTA works for multipath analysis: If enough execution time observations are obtained under each path, the effect that Aj can suffer from any of the Bl in each path is captured in probabilistic terms. This is a sufficient condition for MBPTA to provide safe upper-bounds.

References

1. Sarah Boslaugh and Paul Andrew Watters. *Statistics in a nutshell*. O'Reilly Media, Inc., 2008.
2. J.V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.
3. F.J. Cazorla, E. Qui nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.
4. L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
5. L. Kosmidis, J. Abella, E. Quinones, and F.J. Cazorla. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
6. L. Kosmidis, C. Curtsinger, E. Quinones, J. Abella, E. Berger, and F.J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.
7. L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, and F.J. Cazorla. Achieving timing composability with measurement-based probabilistic timing analysis. In *In IEEE International Symposium on Object/component/service-oriented Real-time distributed computing (ISORC)*, 2013.
8. Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
9. M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.
10. Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
11. Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.