16th International Workshop on Worst-Case Execution Time Analysis

WCET 2016, July 5, 2016, Toulouse, France

Edited by

Martin Schoeberl

Editor

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Lyngby
Denmark
masca@dtu.dk

ACM Classification 1998
B.8.2 Performance Analysis and Design Aids, C.3 Real-Time and Embedded systems, D.2.4 Software/-Program Verification, D.4.7 [Organization and Design] Real-time Systems and Embedded Systems

ISBN 978-3-95977-025-5

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/978-3-95977-025-5.

Publication date
December, 2016

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

Digital Object Identifier: 10.4230/WCET.2016.0

ISBN 978-3-95977-025-5 **ISSN 1868-8969** **http://www.dagstuhl.de/oasics**

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.

ISSN 2190-6807

http://www.dagstuhl.de/oasics

Contents

Regular Papers

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

It is a great pleasure to welcome you to the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016). This year we had 20 paper submitted. Each paper was reviewed by four members of the program committee. From those 20 papers we selected 12 papers for presentation at the workshop and publication in the proceedings. The proceedings of WCET 2016 will be published through the Schloss Dagstuhl's OASIcs online proceedings series, as they were in the last years.

I would like to thank all authors for their contribution to WCET 2016 and all program committee members for their insightful and helpful reviews. This year's WCET workshop received financial support by the EU COST Action IC1202: Timing Analysis on Code-Level (TACLe) and by the COST Office, which is highly appreciated. WCET 2016 is being organized as satellite workshop of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016). I am therefore grateful to the ECRTS 2016 general chair, Christian Fraboul, his local team, and the Real-Time Technical Committee Chair of Euromicro, Gerhard Fohler, for their support.

I hope that you will find this program interesting and maybe triggering new ideas. I wish you informative and exciting sessions and and stimulating discussions during and between the sessions to share ideas with other researchers and practitioners.

Martin Schoeberl

Program Chair, July, 2016

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl

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

List of Authors

Jaume Abella

Irune Agirre

Mikel Azkarate-Askasua

Sebastian Altmeyer

Jean-Luc Béchennec

Florian Brandner

Hugues Cassé

Francisco J Cazorla

Davide Compagnin

Enrique Diaz

Boris Dreyer

Heiko Falk

Sebastien Faucou

Martin Frieb

Peter Hellinckx

Christian Hochberger

Leonidas Kosmidis

Alexander Lange

Bjorn Lisper

Andreas Löfwenmark

Armel Mangean

Enrico Mezzetti

Marianne de Michiel

Briday Mikaël

Jörg Mische

David Morales

Vincent Mussot

Simin Nadjm-Tehrani

Amine Naji

Vincent Nelis

Luis Miguel Pinho

Wolfgang Puffitsch

Eduardo Quiñenones

Christine Rochange

Jordy Ruiz

Martin Schoeberl

Pascal Sotin

Alexander Stegmeier

Rasmus Bo Sørensen

Wei-Tsun Sun

Theo Ungerer

Tullio Vardanega

Peter Waegemann

Simon Wegener

Alexander Weiss

Patrick Meumeu Yomsi

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl

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

Committee

Program Chair

- Martin Schoeberl, Technical University of Denmark

Program Committe

- Sebastian Altmeyer, University of Luxembourg, Luxembourg
- Guillem Bernat, Rapita Systems, UK
- Hugues Casse, IRIT – Universite de Toulouse, France
- Francisco J. Cazorla, Barcelona Supercomputing Center, Spain
- Heiko Falk, TU Hamburg-Harburg, Germany
- Damien Hardy, IRISA, France
- Raimund Kirner, University of Hertfordshire, UK
- Jens Knoop, Vienna University of Technology, Austria
- Bjorn Lisper, University College of Malardalen, Sweden
- Claire Maiza, Grenoble INP/Verimag, France
- Enrico Mezzetti, University of Padua, Italy
- Wolfgang Puffitsch, Technical University of Denmark, Denmark
- Isabelle Puaut, IRISA, France
- Peter Puschner, TU Wien, Austria
- Harini Ramaprasad, University of North Carolina at Charlotte, USA
- Christine Rochange, IRIT, France
- Martin Schoeberl, Technical University of Denmark, Denmark
- Tullio Vardanega, University of Padua, Italy

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl

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

Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis*

Enrique Díaz[1], Jaume Abella[2], Enrico Mezzetti[3], Irune Agirre[4], Mikel Azkarate-Askasua[5], Tullio Vardanega[6], and Francisco J. Cazorla[7]

1 **Universitat Politècnica de Catalunya, Barcelona, Spain; and Barcelona Supercomputing Center, Barcelona, Spain**
2 **Barcelona Supercomputing Center, Barcelona, Spain**
3 **Barcelona Supercomputing Center, Barcelona, Spain**
4 **IK4-IKERLAN, Spain**
5 **IK4-IKERLAN, Spain**
6 **University of Padova, Padova, Italy**
7 **Barcelona Supercomputing Center, Barcelona, Spain; and IIIA-CSIC, IIIA-CSIC, Spain**

Abstract

Measurement-based timing analysis (MBTA) is often used to determine the timing behaviour of software programs embedded in safety-aware real-time systems deployed in various industrial domains including automotive and railway. MBTA methods rely on some form of instrumentation, either at hardware or software level, of the target program or fragments thereof to collect execution-time measurement data. A known drawback of software-level instrumentation is that instrumentation itself does affect the timing and functional behaviour of a program, resulting in the so-called *probe effect*: leaving the instrumentation code in the final executable can negatively affect average performance and could not be even admissible under stringent industrial qualification and certification standards; removing it before operation jeopardizes the results of timing analysis as the WCET estimates on the instrumented version of the program cannot be valid any more due, for example, to the timing effects incurred by different cache alignments. In this paper, we present a novel approach to mitigate the impact of instrumentation code on cache behaviour by reducing the instrumentation overhead while at the same time preserving and consolidating the results of timing analysis.

1998 ACM Subject Classification D.4.7 Real-time Systems and Embedded Systems

Keywords and phrases WCET, Measurements, Instrumentation overhead

Digital Object Identifier 10.4230/OASIcs.WCET.2016.1

1 Introduction

Measurement-based timing analysis (MBTA) methods are widely used in application domains such as automotive, railway or space [19]. With MBTA, execution-time measurements of

* The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (`http://www.proxima-project.eu`), grant agreement no. 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation (grant TIN2015-65316-P) and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship RYC-2013-14717.

selected fragments of software programs of interest are taken while observing runs of the program on the target processor, performed under stressful conditions. The highest value, usually known as high-water mark time (HWMT), is recorded. The HWMT of all code fragments (the smallest of which is a basic block) in the program are then combined to determine the worst-case execution time (WCET) of the corresponding task. It is worth noting that MBTA works on the premise that the testing conditions are representative of real system operation, so that the HWMT approximates the real WCET.

The places in the code where the required execution-time observations are made are usually referred to as *instrumentation points* (ipoints). MBTA generates a run-time trace that logs which ipoints are traversed at what time during the observation run. Two differing approaches exist to generate time readings at ipoints: the generation of time traces via hardware methods has been made possible with the advent of processors with advanced debug capabilities that do not affect program timing behaviour. Hardware instrumentation ideally provides transparent generation of timing traces – assuming that collected traces can be output in a fast-enough and equally transparent way so that no trace data are lost because of overfull buffers and no explicit program action is to be taken. However, this support is not present in all processors candidate for use in real-time systems. As a result, in the general case, software-level solutions are adopted where some form of instrumentation code is required in the program to generate timing information.

In contrast with its hardware-based counterpart software instrumentation is highly intrusive, especially on the temporal behaviour. In the presence of caches, instrumentation code can generate unwanted timing effects far beyond the intrinsic timing penalty of the additional instrumentation instructions, that would not have been observed in the un-instrumented (original) program. The user thus faces the dilemma of whether to remove the instrumentation code from or leave it in the final program.

- Removing the instrumentation code from the final executable raises the question of how the execution-time observations taken with the instrumented code correlate with the timing behaviour of the un-instrumented program. In fact, both functional and timing verification would have been conducted on a different software artifact and strong additional argument must be provided for the analysis result to hold.
- Leaving instrumentation code in the final executable spares the burden (for cost and complexity) of demonstrating equivalent functionality as WCET estimation is performed on the executable that will be deployed in the operational system. However, certification and qualification practices may simply not accept the presence of this instrumenter-added code in the executable. As an immediate effect, leaving instrumentation code in a program is likely to worsen memory footprint and average performance. Further, some memory-mapped I/O space – where execution-time readings might be kept – may be unnecessarily wasted.

In fact, both leaving and removing instrumentation code may have disruptive effects on the certification process. While there have been several methods to minimise the number of instrumentation points, without losing too much precision [17], in this paper we show that even a single instrumentation point can lead to significantly different timing behaviour between the original (un-instrumented) program and the instrumented version.

We present a novel technique that strikes an optimal balance between the two approaches, basing on the concept of *functionally-neutral* program, that is used at system operation. From the original program (oprog), fnprog is generated by inserting *nop* instructions at desired instrumentation points. The neutral nature of nop instructions and the fact that they can neither generate interrupts nor have input or output dependences simplifies certi-

fication/qualification argumentation. For the purposes of timing analysis, nop instructions are replaced by actual instrumentation operations, resulting in an instrumented program (iprog). The number of nops inserted per ipoint in fnprog is carefully selected so that the cache alignment of code in fnprog and iprog stays unchanged. This prevents any unwanted impact on timing behaviour that may stem from variations in cache alignment. The increase in terms of memory footprint and execution time of fnprog as compared to oprog, instead, depends on the program and the number of ipoints inserted.

We have applied this method within the scope of measurement-based probabilistic timing analysis (MBPTA) [2]. Besides its evident benefits on the fronts of qualification and certification alike, our approach significantly reduces the impact of software-level instrumentation. In quantitative terms, assuming basic block level instrumentation with two instructions per ipoint, the average degradation we observed in the computed execution-time bounds was 9.3% for EEMBC benchmark programs and 8.7% for a railway case-study application.

2 Background and Problem Statement

MBTA differentiates between the *analysis phase*, when verification of timing behaviour takes place, and the *operation phase*, when the system is deployed into operation. MBTA computes WCET estimates with execution-time measurements taken at analysis time, on the condition that the corresponding bound holds at operation. This requires the user to define test scenarios that trigger worst-case conditions that can occur during system operation. (We intentionally omit discussing here how difficult, if at all possible, that is for the user.)

MBPTA [2] is a variant of MBTA that derives probabilistic WCET (pWCET) estimates – an execution-time distribution expressing the maximum probability that upper bounds the residual risk that *one instance* of the program may exceed a given execution-time threshold. MBPTA handles the sources of execution-time variability caused by hardware and software effects by ensuring that the jitter they cause at analysis matches or upper bounds that which occurs during operation [8]. Cache memories, whose behaviour cannot be treated that way, are time randomised instead [6], so that their impact on execution time can be studied probabilistically for both analysis and operation conditions and the former can be made to upper bound the latter. MBPTA employs extreme value theory [9] (EVT) to model the distribution of extreme (worst-case) execution times (pWCET), considering the timing impact of randomised hardware resources both individually and collectively.

2.1 Problem Statement

MB(P)TA generates a trace that records which and when *ipoints* are traversed during execution. Every such trace is a sequence of <*ipointid*, *timestamp*> pairs. Two main steps take place in the generation of ipoint traces.

- **Generation**. Modern hardware comprises advanced debug interfaces that trigger specific actions when certain opcodes are executed. For example, debug hardware can be used, on every branch instruction taken by program execution, to collect a <branch (instruction) address, execution cycle> pair of trace data. The type of instruction (event) to trace and the action to perform when such an instruction is hit can be programmed through a provided interface, e.g. Nexus or GRMON for the LEON processor family [15]. Debug hardware of that kind is not present in all processors used in real-time systems. In general, therefore, some form of software instrumentation is needed. To that end, specific instrumentation instructions are inserted at the desired granularity of information in the

execution context of the program of interest. For instance, these instructions may read the time-base register and output its contents to a specific I/O address.

- **Collection**. Once instrumentation (in either hardware or software) is in place, the execution of the unit of analysis on the target processor results in a set of timestamps and events. This list is output outside of the processor and dispatched to some off-line analysis tool. The capture and offloading process can be the source of further interference. On-chip debug hardware usually includes off-band buses so that the transfer of <timestamp, event> pairs does not affect the execution of the unit of analysis. Depending on the speed of the CPU and the quantity of ipoints, the volume of timestamped data can be high. This can be managed by outputting the data to high-speed ports. Specialised hardware to process the trace at very high speed has been proposed [12] and exists commercially [14], which prevents loss of information or stalls of execution.

The generation and collection of this trace may interfere with the system's timing and perturb its behaviour. This phenomenon is known as the *probe effect*, which causes the instrumented program to have register and cache usage profiles that differ from those of the original (un-instrumented) program.

We categorise those effects as follows:

- **Direct impact** stems from the execution latency of fetching and executing instrumentation code (icode). The icode usually involves reading internal processor registers, e.g., the time register, ($\Delta_{icode}^{core-exec}$) and outputting the readout to a specific memory address. If this information is transferred via buses or I/O controllers used by the application under analysis, this action is bound to interfere with application's timing behaviour. Furthermore, if ipoint information is cached, this can also cause significant effects ($\Delta_{icode}^{chip-exec}$). Not all processors are capable of tracing and dumping data implicitly, without using ipoints. Fast debug links and I/O ports (e.g., GPIO, Ethernet) are often available to dump trace data transparently to the application as long as the ipoints are conveniently placed in the program being run. Trace data can be either processed in real-time of stored for off-line processing.
- **Indirect impact** of the icode arises from the change in the layout of program code in memory. When an ipoint is inserted in the program, it shifts the position in memory of the subsequent instructions and hence also their address and possibly its cache set layout. This may make it considerably difficult for the user to provide evidence that the execution-time measurements obtained with the instrumented binary (iprog) are larger or smaller than those obtained with oprog. This in turn causes the pWCET estimates obtained for iprog to not safely upper bound oprog's execution time.

The execution time of the original program ($ETo prog$) is affected in a direct manner by icode in the instrumented program ($ETi prog$) as shown in the second addend of Equation 1.

$$ETiprog = ETo prog + \left(\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect}\right) + \Delta_{icode}^{malign} \quad (1)$$

Without loss of generality, we assume that both trace-generation impact, other than executing icode at the core level, and trace-collection overhead are null ($\Delta_{icode}^{collect} = \Delta_{icode}^{chip-exec} = 0$), and instrumentation overheads only arise from the core execution of ipoints included in the unit of analysis ($\Delta_{icode}^{core-exec} \neq 0$), which however creates a constant execution time overhead. The actual problem stems from the fact that the insertion of the icode may change the memory layout of the original code in oprog. Unlike the direct impact of icode that is bound to be a positive value, misalignment impact (Δ_{icode}^{malign}) can be either positive or negative. This may cause iprog to run either faster or slower than oprog. This may lead to the situation

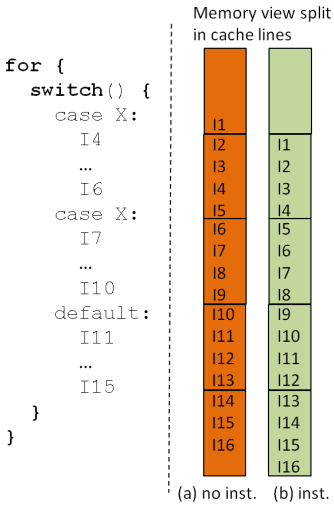


Figure 1 Memory impact.

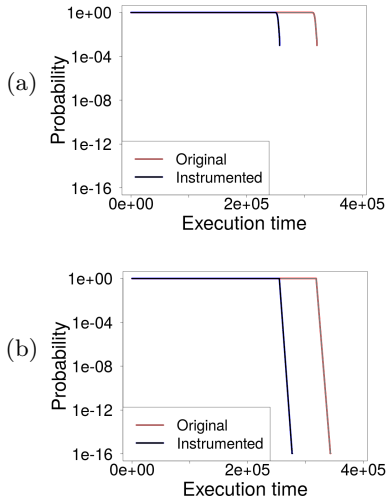


Figure 2 Execution-time impact.

in which, despite the direct impact caused by the insertion of icode, the execution-time distribution and pWCET estimate for iprog are even smaller than those for obtained for oprog, i.e. $\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect} < \Delta_{icode}^{malign}$ so $\Delta_{icode}^{core-exec} < \Delta_{icode}^{malign}$.

2.2 Illustrative Example

In this section we use a small example to demonstrate how instrumentation code can create unexpected timing behaviour in which the instrumented program yields an execution-time profile that does not upper bound the profile of the un-instrumented program.

Let us consider the code fragment shown in the left part of Figure 1. This code comprises a `for` structure with a nested `switch` structure. In the example, $I1$, $I2$ and $I3$ (not shown) correspond to loop and switch control instructions; $I4 - I6$ are in the first case of the switch; $I7 - I10$ in the second; and $I11 - I15$ in the third. Further assume that before any instrumentation is applied the code (instruction addresses) is laid out in memory as presented in (a), occupying 5 cache lines. Further assume that a single instrumentation instruction is inserted somewhere before this fragment, causing a shift of one instruction and resulting in the memory layout presented in (b). The body of the loop thus occupies 4 lines.

We executed this code on a light-weight processor simulator comprising a 2-set 2-way instruction cache. Other than the jitter caused by the instruction cache – 4 cycles in case of hit and 94 cycles in case of miss – instructions have a back-end latency of 6 cycles. We assume an arbitrary input vector that causes a different branch of the `switch` to be triggered at each loop iteration. We assume $\Delta_{icode}^{core-exec} = 2$ for the only added ipoint.

In terms of MBPTA, this yields the empirical complementary cumulative distribution functions (ECCDFs) shown in Figure 2(a) from where we see that the execution profile of the un-instrumented code is higher than that of the instrumented code. If we apply MBPTA to those observed execution times we obtain the pWCET estimates shown in Figure 2(b). We observe that both the empirical distribution and the pWCET for the original code are much higher than those for the instrumented code.

Hence, even a single instrumentation instruction can change the cache layout so that the instrumented program has lower execution time than the un-instrumented one.

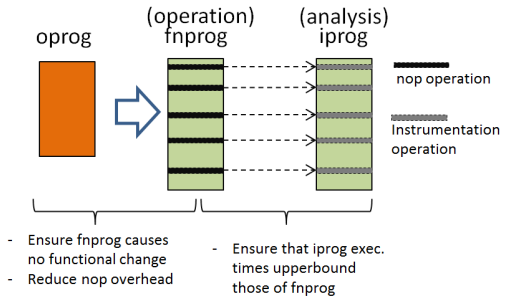


Figure 3 Schematics of the proposed approach.

3 Proposal

The proposal we present in this section aims to help the user be assured that the version of the program used for WCET analysis leads to an execution-time distribution that reliably upper bounds the execution time of the version of the program used during operation. As a secondary goal, we want to reduce $\Delta_{icode}^{core-exec}$ to produce tighter WCET estimates.

Our approach uses three versions of the program of interest, see Figure 3: oprog, the original program, fnprog, its augmented functionally-neutral version, which is used in operation, and iprog, the instrumented executable, which is used for analysis.

fnprog. This is a version of oprog augmented with *nop* instructions inserted to bound the indirect impact of the icode (Δ_{icode}^{malign}). This modification results in a program with unaltered functional behaviour that can therefore be used in operation. However, an argument (which we denote A1) is needed to show that fnprog provides the same functional output as oprog. Furthermore, we also need to prove that the average performance of fnprog is near and not unacceptably worse than that of oprog.

The nop instructions in fnprog are inserted at all places where ipoints are needed. The number of nop instructions inserted per ipoint is sufficient to ensure that, when the actual instrumentation instructions are inserted in iprog, no cache-line misalignment occurs. Typically, one or two instrumentation instructions per ipoint are sufficient to collect timing information, depending on the actual hardware support and instruction set. Hence, fnprog includes one or two nops at the place of each ipoint. It is worth noting that oprog may already include some nops. For instance, in an architecture with delayed branches (e.g., SparcV8), delayed slots may not be filled with useful instructions or nops, depending on compiler flags or program semantics. Further, some compilers include options that insert nops to enforce memory alignment at the level of function, branches, jumps and loops (e.g. `-falign-functions=n`, `-falign-labels=n`, `-falign-loops=n`, `-falign-jumps=n` in GCC). Those nops are just fine for placing instrumentation code in iprog.

iprog. The execution-time measurement traces needed by MB(P)TA are obtained from a modified version of fnprog. In that version, which we call instrumented binary, iprog, some or even all of the inserted nops are replaced by actual instrumentation instructions impacting execution time. This change is made in a way that causes no code realignment with respect to fnprog which simplifies ensuring that the execution-time traces obtained from iprog can be reliably used to derive a pWCET estimate for fnprog as used in operation. A further argument (A2) is needed to show that the execution-time behaviour of iprog is never less

than that of fnprog, so that the execution-time observations taken for iprog can be used to upper bound the WCET of fnprog.

If our approach is adopted, the required nops could be automatically added by the (qualified) compiler. The number of nops and the level they are added could be easily controlled via compiler parameters, e.g. `-fnopcount=n` and `-fnoplevel=basicblock`.

3.1 Functionally-neutral impact of fnprog (A1)

The use of nops simplifies providing arguments that fnprog does not change the functional behaviour of oprog: most of today's Instruction Set Architectures (ISAs) include *nop*-type instructions, whose function is to perform no operation in the processor other than fetch and, possibly, decode, where the instruction is usually stripped from the execution stream.

The main advantage of using nops is that they are functionally neutral: (1) by definition, a nop performs no operation; (2) its execution does not change status flags or any other control registers; (3) a nop generates neither raises interrupts nor exceptions; (4) a nop uses no architectural (programmer accessible) register, which allows inserting nops anywhere in the code; and (5) a nop has no input and no output (register) dependences. From all these properties it follows that fnprog cannot change the functional behaviour of oprog.

From the average performance standpoint, whose improvement we defined as our second goal, nops usually take a few cycles to execute. In some architectures, the processor may even strip nops out from the pipeline before they reach the execution stage. This is in contrast with actual instrumentation instructions that usually need to access off-core (or off-chip) resources such as I/O ports or trace buffers, thus incurring longer execution times.

3.2 fnprog's WCET is upper bounded by that of iprog (A2)

The difference between fnprog and iprog is that some nops in the former are changed to actual instrumentation instructions in the latter. The number of nops inserted at the place of each ipoint in fnprog is such that *exactly* the same cache-line alignment occurs in fnprog and in iprog. The net result is an increase in the execution time of iprog in comparison to fnprog since the instrumentation code takes longer to execute than nops. The fact that this overhead has an additive nature ipoint by ipoint – in a processor free of timing anomalies – facilitates making an argument about the fact that the resulting execution-time distribution collected with iprog upper bounds that of fnprog. Notably, the execution time of fnprog might be lower than that of oprog, owing to the instruction cache effects described before. However, this does not create any issue since fnprog is the one that will be deployed to operation.

Recent work [7] in the specific context of MBPTA [2] for time-randomised caches [6] helps us contend that the execution time of iprog does indeed upper bound that of fnprog. Let IS_{org} be a sequence of instructions to which we add a set $\mathcal{O}$ of new operations – both acting within core (e.g. add) and on memory – resulting in the extended instruction sequence IS_{ext}.

The authors of [7] show that the probabilistic execution time (pET) of IS_{ext} is higher than the pET of IS_{org}. We say that $pET(ISi) \geq pET(ISj)$ if, for any cut-off probability, the execution time of ISi is higher than or equal to the execution time of ISj. We contend that this argument can be made for standard MBTA, not just MBPTA, but we leave the corresponding demonstration as future work.

In addition to cache-related icode variability – which we attack in this paper – measurement-based timing analysis needs to handle timing anomalies if they can happen.

Table 1 Average increase in code size and execution time of fnprog and iprog normalized to oprog.

no. instruct.	Code Size	execution time fnprog	execution time iprog
1 inst./2 inst.	6.87%/13.74%	4.35%/9.28%	8.33%/17.54%

4 Experimental Results

So far we have provided arguments in support to the fact that our functionally-neutral approach provides a reasonable solution to the software instrumentation problem from the qualification and certification standpoint as fnprog can be safely deployed instead of the oprog and iprog. In our experimental evaluation we aimed at providing evidence that the fnprog always exhibits pWCET that tightly upper bounds oprog and is always lower than iprog. For our evaluation we use the well-known EEMBC automotive benchmark suite [11]. In particular, we use the following benchmark programs: `a2time` (A2), `aifftr` (AI), `aifirf` (AF), `aiifft` (AT), `bitmnp` (BI),`cacheb` (CB), `canrdr` (CN), `idctrn` (ID), `iirflt` (II), `matrix` (MA). We also use a railway case-study application that is part of the European Railway Traffic Management System (ERTMS) [4] initiative that seeks to define a unique European train signalling standard. Our focus is on the on-board unit of the ERTMS, called European Train Control System (ETCS). We consider 10 different input sets (`S0` to `S9`). For the iprog we assume 1 and 2 instrumentation instructions per ipoint.

We focus on the pessimistic scenario where ipoints are added at basic block boundary – the smallest granularity of instrumentation in general. While in this case the instrumentation impact is high, we have seen that a single instrumentation point can cause unwanted cache effects between the instrumented and non-instrumented code. In the presence of techniques reducing the number of ipoints (see Related Work Section), the overhead introduced by our approach will naturally reduce. We use a cycle-accurate processor simulator that implements 4KB L1 instruction- and data-caches, which comprise 128 sets and 2 ways each. Both caches implement random placement and replacement [6]. The access latency to the L1 caches is 1 cycle and that to main memory is 28 cycles. For the instrumentation instructions, we assume they have the cost of 2 cycles.

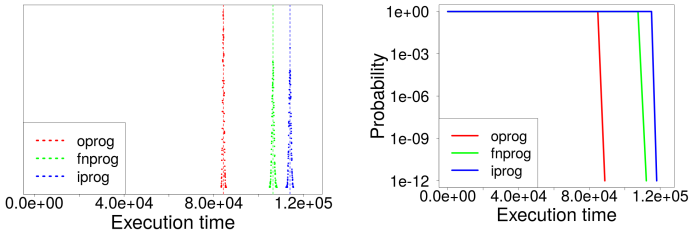
4.1 Execution Time and Code Size Increase Due to Instrumentation

Table 1 shows the increase in code size and execution time incurred by fnprog and iprog respectively. We can see that fnprog incurs moderate overhead in footprint – between 7% and 14% – when one or two nop operations are added per ipoint per basic block. In terms of execution time, the impact is 5% and 10% when 1 and 2 instructions are added respectively. Meanwhile iprog incurs higher execution time overheads: 9% and 19% respectively.

Per-benchmark results (not depicted for space constraints), show a clear relation between program's average basic block size the the code size impact of nops. In terms of time, in addition to average basic block size the average duration of each instruction (program's CPI) determines the execution overhead of nops.

4.2 pWCET estimates

For the case of 2 instrumentation operations per ipoint, we compare the pWCET estimates obtained from the execution-time measurements from oprog, fnprog and iprog. We collected 1,000 runs for each version, which have sufficed to pass the MBPTA convergence



(a) Histogram (b) pWCET estimate

Figure 4 Histogram and MBPTA projection for `a2time`.

Table 2 pWCET estimates for 10^{-12} for fnprog and iprog normalized to oprog.

prog.	A2	AI	AF	AT	BF	BI	CB	CN	ID	II	MA
fnprog	26.4%	3.8%	11.6%	7.1%	11.5%	11.9%	2.1%	14.1%	12.3%	11.9%	6.4%
iprog	33.0%	9.3%	22.1%	12.4%	22.0%	16.8%	4.0%	27.1%	23.3%	21.6%	12.7%

criteria [2]. Figure 4 shows the execution-time histogram and the resulting pWCET estimate for `a2time` benchmark. We observe how fnprog is close to oprog's. Further, changing nops by instrumentation instructions causes iprog's execution-time to upper bound fnprog's.

For the other benchmarks, Table 2 summarises the difference observed for the three versions of the program for a cut-off probability of 10^{-12} per run. As shown, fnprog is relatively tight with respect to oprog. The only exception is `A2`, which includes many basic blocks, and therefore takes a higher density of nops.

4.3 Railway case study

Table 3 reports analogous results for the railway case study for 2 instrumentation instructions per ipoint. The results are even tighter on average than those we obtained for the EEMBC benchmarks, with an average increase in pWCET estimates of 8.7% and 11.9% for fnprog and iprog respectively. The code size increase observed is 12%, which is also less than the average incurred with the EEMBC benchmarks. Overall, our proposed solution provably abates the negative misalignment effects of icode, for a small cost in execution time in general.

5 Related Work

The literature on measurement-based timing analysis is abundant [17, 10, 16, 17, 1, 19].

In [10], the authors discuss the pros and cons of several ways to collect execution traces and how the frequency of ipoints results in light-weight or heavy-weight instrumentation.

Measurements can be taken (i.e., ipoints can be placed) at program boundaries. However, hybrid mechanisms exist to identify smaller program parts. The particular segments used are extracted from an analysis of the Control-Flow Graph [5, 1]. The segments (partitions) are chosen to facilitate the derivation of a WCET by composing the WCET of each segment, facilitating measurements [17, 1] or reducing the number of ipoints. Some of these techniques also work on an automatic generation of input data [5, 17, 18]. In [5], the authors decompose execution paths into sub-paths and then use formal methods to derive the required test data (in an automatic manner) and measure the execution time of the sub-paths.

Table 3 pWCET estimates for 10^{-12} for *fnprog* and *iprog* w.r.t. that for *oprog*.

prog.	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
fnprog	8.4%	7.6%	9.5%	9.1%	8.1%	9.5%	9.4%	8.3%	8.6%	8.9%
iprog	11.5%	10.4%	12.1%	12.3%	12.2%	13.3%	12.4%	11.9%	12.2%	11.1%

In [13], the authors propose the concept of context-sensitive traces to capture the impact of execution history on the precision of measurement-based execution-time estimates. This is done using the concept of "call string" that defines the sequence (of the last k calls) that keeps information similar to the call stack. For trace collection, some research approaches developed an FPGA board to transfer information from the target to the host to increase trace processing capabilities [12]. Other approaches propose on-line aggregation of timing data removing the need for collecting and post-processing long traces [3].

6 Conclusions and Future Work

We have presented a new approach to mitigate the impact of instrumentation code to prevent cache misalignments from occurring between the instrumented and un-instrumented versions of the program under analysis, while incurring low overhead in terms of execution time. In particular, we build upon the use of *functionally-neutral* operations such as nops to create a program version to be deployed that is functionally equivalent to the original program, and has a provable lower execution time than the instrumented version.

As part of our future work we plan to evaluate the fnprog approach in a real hardware platform and a commercial timing analysis tool. We also plan to extend the argumentation about the timing impact of iprog w.r.t. fnprog to non probabilistic timing analysis.

References

1 Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 54–63, 2010. doi:10.4230/OASIcs.WCET.2010.54.

2 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 91–101, 2012. doi:10.1109/ECRTS.2012.31.

3 Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise continuous non-intrusive measurement-based execution time estimation. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 45–54, 2015. doi:10.4230/OASIcs.WCET.2015.45.

4 ERA (European Railway Agency). ERTMS – Set of specifications – 2 (ETCS baseline 3 and GSM-R baseline 0), 2014. URL: http://www.era.europa.eu/Core-Activities/ERTMS/Pages/Set-of-specifications-2.aspx.

5 Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *4th International Workshop on Worst-Case Execution Time Analysis, WCET 2004, Catania, Sicily, Italy, June 29, 2004*, pages 67–70, 2004. URL: https://www.irisa.fr/manifestations/2004/wcet2004/.

6 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 513–518, 2013. doi:10.7873/DATE.2013.116.

7 Leonidas Kosmidis, Jaume Abella, Franck Wartel, Eduardo Quiñones, Antoine Colin, and Francisco J. Cazorla. PUB: path upper-bounding for measurement-based probabilistic timing analysis. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 276–287, 2014. doi:10.1109/ECRTS.2014.34.

8 Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, Ian Broster, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*, pages 401–410, 2014. doi:10.1109/DSD.2014.50.

9 Samuel Kotz and Saralees Nadarajah. *Extreme Value Distributions: Theory and Applications*. World Scientific, 2000. ISBN 978-1-86094-224-2.

10 Stefan M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 – a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, pages 75–78, 2003.

11 Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009. doi:10.1109/MM.2009.74.

12 Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, and Peter P. Puschner. Using a runtime measurement device with measurement-based WCET analysis. In *Embedded System Design: Topics, Techniques and Trends, IFIP TC10 Working Conference: International Embedded Systems Symposium (IESS), May 30 - June 1, 2007, Irvine, CA, USA*, pages 15–26, 2007. doi:10.1007/978-0-387-72258-0_2.

13 Stefan Stattelmann and Florian Martin. On the use of context information for precise measurement-based execution time estimation. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 64–76, 2010. doi:10.4230/OASIcs.WCET.2010.64.

14 https://www.rapitasystems.com/products/rtbx. *RTBx*. Rapita Systems.

15 http://www.gaisler.com/index.php/products/debug-tools/grmon. *Cobham Gaisler*. GRMON.

16 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based worst-case execution time analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, SEUS 2005, Seattle, WA, USA, May 16-17, 2005*, pages 7–10, 2005. doi:10.1109/SEUS.2005.12.

17 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, pages 430–444, 2008. doi:10.1007/978-3-540-88479-8_30.

18 Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 606–611, 2005. doi:10.1109/DATE.2005.76.

19 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.

TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research*

Heiko Falk[1], Sebastian Altmeyer[2], Peter Hellinckx[3], Björn Lisper[4], Wolfgang Puffitsch[5], Christine Rochange[6], Martin Schoeberl[7], Rasmus Bo Sørensen[8], Peter Wägemann[9], and Simon Wegener[10]

1 **Hamburg University of Technology, Institute of Embedded Systems, Hamburg, Germany**
Heiko.Falk@tuhh.de
2 **University of Amsterdam, Amsterdam, The Netherlands**
altmeyer@uva.nl
3 **University of Antwerp, iMinds, Antwerp, Belgium**
peter.hellinckx@uantwerpen.be
4 **Mälardalen University, School of Innovation, Design, and Engineering, Västerås, Sweden**
bjorn.lisper@mdh.se
5 **Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark**
wopu@dtu.dk
6 **University of Toulouse, Toulouse, France**
rochange@irit.fr
7 **Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark**
masca@dtu.dk
8 **Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark**
rboso@dtu.dk
9 **Friedrich-Alexander University Erlangen-Nürnberg, Erlangen-Nürnberg, Germany**
waegemann@cs.fau.de
10 **AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany**
wegener@absint.com

Abstract

Engineering related research, such as research on worst-case execution time, uses experimentation to evaluate ideas. For these experiments we need example programs. Furthermore, to make the research experimentation repeatable those programs shall be made publicly available.

We collected open-source programs, adapted them to a common coding style, and provide the collection in open-source. The benchmark collection is called TACLeBench and is available from GitHub in version 1.9 at the publication date of this paper. One of the main features of TACLeBench is that all programs are self-contained without any dependencies on standard libraries or an operating system.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems

Keywords and phrases Benchmark, WCET analysis, real-time systems

Digital Object Identifier 10.4230/OASIcs.WCET.2016.2

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

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl; Article No. 2; pp. 2:1–2:10

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

1 Introduction

Good, realistic benchmark suites are essential for the evaluation and comparison of worst-case execution time (WCET) analysis, compiler, and computer architecture techniques. TACLeBench provides a freely available and comprehensive benchmark suite for timing analysis and related research topics. TACLeBench will be continuously extended by novel benchmarks, especially by parallel multi-task/multi-core benchmarks. The extension of TACLeBench will be carefully managed with snapshots and versioning so that it is clear which code has been used in a research experiment. The overall goal is to establish TACLeBench as the standard benchmarking suite for WCET analysis, WCET oriented compiler and computer architecture research worldwide.

TACLeBench is a collection of currently 53 benchmark programs from several different research groups and tool vendors around the world. These benchmarks are provided as ISO C99 source codes. The source codes are 100% self-contained; no dependencies to system-specific header files via `#include` directives or an operating system exist. All input data is part of the C source code. Potentially used functions from math libraries are also provided in the form of C source code. This makes the TACLeBench collection useful for general embedded/barebone systems where no standard library is available.

Furthermore, almost all benchmarks are processor-independent and can be compiled and evaluated for any kind of target processor. The only exception is PapaBench that uses architecture-dependent I/O addresses and currently supports Atmel AVR processors only.

Since TACLeBench addresses the needs imposed by timing analysis tools, all benchmarks are completely annotated with flow facts. These flow facts are directly incorporated into the C source codes using pragmas. TACLeBench distinguishes between so-called flow restrictions, loop bounds, and entry points. Besides flow restrictions, TACLeBench also uses loop bound flow facts, which simplify the annotation of regular loops. Loop bounds provide an upper and a lower bound for the number of iterations of the annotated loop. Finally, TACLeBench uses entry point annotations that denote points in a program's control flow graph where the control flow may start. Typically, this is the "main" function of a program, but in a (possibly interrupt-driven) multi-task system, there may be multiple entry points in a single set of source files. These entry points may even share some common code. In order to mark such task entries, each function of a multi-task application where a task begins can be marked as an entry point. The complete specification of the used flow fact language can be found in [3], which is part of the source distribution.

If you would like to share your benchmarks with us, feel free to contact Heiko Falk (or any coauthor of this paper) in order to have your source codes included in TACLeBench.

The first version of TACLeBench (version 1.0, available from[1]), which was produced by Heiko Falk, was a collection of 102 programs from several different research groups. We keep this first version tagged with "V1.0" in the public GitHub repository.[2] The version described in this paper is version 1.9 and tagged as such in the repository. Version 2.0 will be soon available when the last missing programs have been formatted. The intention is to have the HEAD of the master branch being the most recent, versioned snapshot of TACLeBench. Future development and additions will be performed on a development branch so that HEAD of master is always a consistent snapshot.

This paper is organized in 5 sections: The following section presents related work.

[1] `http://www.tacle.eu/index.php/activities/taclebench`
[2] `https://github.com/tacle/tacle-bench`

Section 3 presents the benchmark collection, its classification, and the updates to make them useful. Section 4 evaluates the benchmark collection. Section 5 concludes.

2 Related Work

The Mälardalen WCET benchmarks (MRTC) [6] is the first collection of programs especially intended for benchmarking WCET analysis tools, with a focus on program flow analysis. It was collected from several sources in 2005, and has since then been used in many WCET research projects as well as for the WCET Tool Challenge 2006 [5]. A subset of the Mälardalen benchmarks has even been translated to Java [8]. Most benchmarks are relatively small, except two C programs that have been generated from tools. The benchmarks also contain all input data. This effectively turns them into single-path programs, which makes them less suitable for evaluating tools that can handle multi-path codes. We include most of the benchmarks from the Mälardalen WCET benchmark suite in TACLeBench. We dropped benchmarks where the licensing terms are unknown or even disallow distributing the source.

MiBench [7] is a collection of benchmarks targeting the embedded domain and providing them in open-source. We include some of the MiBench benchmarks, especially those where it was possible to include the input data with the C source.

DEBIE [9] is a program derived from a satellite-mounted detector of micro-meteorids and space debris. It was developed by Space Systems Finland Ltd and was converted by Tidorum Ltd into a portable benchmark for real-time applications. DEBIE is a multi-task application that consists of 8, partially small, different tasks. DEBIE is accompanied by a specification of valid input and output data and of required activation rates of the individual tasks.

PapaBench [10] has been derived from Paparazzi,[3] a project for unmanned aerial vehicles. PapaBench includes two software components that run on separate processors: the *fly-by-wire* part controls the flight while the *autopilot* part controls the GPS and executes the flight plan (which is decided offline). Both software parts cumulate 13 tasks that are subject to precedence constraints and 6 interrupt service routines.

The Embedded Microprocessor Benchmark Consortium (EEMBC) [2] provides a benchmark suite dedicated to the evaluation of the performance of embedded hardware and embedded software. The benchmarks are divided into subsets according to the target domain, e.g., the automotive domain, phones and tablets, but also big data and cloud computing. To improve comparability between different systems, the consortium provides a test-harness that allows deriving certifiable scores. The test-harness, being a clear advantage in terms of comparability, constitutes a hindrance in terms of portability and usability. Whereas the TACLeBench has been designed to ease portability and to allow the immediate use of the benchmark with a large variety of tools and platforms, the EEMBC benchmarks are not stand-alone executable without the test-harness. Furthermore, in stark contrast to TACLeBench, the EEMBC benchmarks are not published under an open-source license. Instead, the benchmarks are behind a pay-wall, even for purely academic research.

JemBench [13] is a Java benchmark suite targeting embedded Java platforms. JemBench only assumes the availability of a CLDC API, the minimal configuration defined for the J2ME. The core of the benchmark suite consists of adapted real-world applications. The benchmarks are structured in *micro*, *kernel*, *application*, *parallel*, and *streaming* benchmarks. Micro benchmarks are used to measure short bytecode sequences; kernel benchmarks compute a computational kernel; and application benchmarks are real-world programs restructured

[3] `https://wiki.paparazziuav.org/`

Table 1 TACLeBench kernel benchmarks.

Name	Description	Code Size (SLOC)	Origin
binarysearch	Binary search of 15 integers	47	SNU-RT
bitcount	Couting number of bits in an integer array	164	Bob Stout & Auke Reitsma
bitonic	Bitonic sorting network	52	MiBench
bsort	Bubblesort program	32	MRTC
complex_updates	Multiply-add with complex vectors	18	DSPStone
countnegative	Counts signes in a matrix	35	MRTC
fac	Factorial function	21	MRTC
fft	1024-point FFT, 13 bits per twiddle	78	DSPStone
filterbank	Filter bank for multirate signals	75	StreamIt
fir2dim	2-dimensional FIR filter convolution	75	DSPStone
iir	Biquad IIR 4 sections filter	27	DSPStone
insertsort	Insertion sort	35	SNU-RT
jfdctint	Discrete-cosine transformation on a 8x8 pixel block	123	SNU-RT
lms	LMS adaptive signal enhancement	51	SNU-RT
ludcmp	LU decomposition	68	SNU-RT
matrix1	Generic matrix multiplication	28	DSPStone
md5	Message digest algorithm	344	NetBench
minver	Floating point matrix inversion	141	SNU-RT
pm	Pattern match kernel	484	HPEC
prime	Prime number test	41	MRTC
quicksort	Quick sort of strings and vectors	992	MiBench
recursion	Artificial recursive code	18	MRTC
sha	NIST secure hash algorithm	382	MiBench
st	Statistics calculations	90	MRTC

as standalone benchmarks. Parallel and streaming benchmarks are intended to explore multicore speedup. Benedikt Huber ported one of the application benchmarks (Lift) to C and we include it in TACLeBench.

3 The Benchmark Collection

3.1 Benchmark Sources and Classification

The benchmarks included in TACLeBench are sourced from single sources and benchmarks collections. The benchmarks are: SNU-RT benchmark suite, MiBench embedded benchmark suite, Mälardalen Real-Time Research Center (MRTC) WCET benchmarks, DSPStone from RWTH Aachen, StreamIt from MIT, NetBench from UCLA, MediaBench, and the HEPC challenge benchmark suite. We have specified the origin of each benchmark in Tables 1–5.

As a measure of the size of each benchmark, we present the number of source lines of code (SLOC). The SLOC count excludes input data arrays and the initialization code. We used the Linux utility `sloccount` to measure the SLOC. The benchmarks are divided into five classes: Kernel, sequential, application, test, and parallel.

Table 2 TACLeBench sequential benchmarks.

Name	Description	Code Size (SLOC)	Origin
adpcm_dec	ADPCM decoder	293	SNU-RT
adpcm_enc	ADPCM encoder	316	SNU-RT
ammunition	C compiler arithmetic stress test	2431	Vladimir Makarov
anagram	Word anagram computation	2710	Raymond Chen
audiobeam	Audio beam former	833	StreamIt
cjpeg_transupp	JPEG image transcoding routines	608	MediaBench
cjpeg_wrbmp	JPEG image bitmap writing code	892	Thomas G. Lane
dijkstra	All pairs shortest path	117	MiBench
epic	Efficient pyramid image coder	451	MediaBench
fmref	Software FM radio with equalizer	680	StreamIt
g723_enc	CCITT G.723 encoder	480	SUN Microsystems
gsm_dec	GSM provisional standard decoder	543	MediaBench
gsm_enc	GSM provisional standard encoder	1491	MediaBench
h264_dec	H.264 block decoding functions	460	MediaBench
huff_dec	Huffman decoding with a file source to decompress	183	David Bourgin
huff_enc	Huffman encoding with a file source to compress	325	David Bourgin
mpeg2	MPEG2 motion estimation	1297	MediaBench
ndes	Complex embedded code	260	MRTC
petrinet	Petri net simulation	500	Friedhelm Stappert
rijndael_dec	Rijndael AES decryption	820	MiBench
rijndael_enc	Rijndael AES encryption	734	MiBench
statemate	Statechart simulation of a car window lift control	1038	Friedhelm Stappert
susan	MR image recognition algorithm	1491	MiBench

The kernel benchmarks, listed in Table 1, are synthetic benchmarks implementing small kernel functions; the size of the kernel benchmarks is in the range of 18 to 992 SLOC.

The sequential benchmarks, listed in Table 2, implement large function blocks, such as encoders and decoders, which are used in many embedded systems. The size of the sequential benchmarks is in the range of 117 to 2710 SLOC. The sequential benchmarks cover graph search, cryptographic algorithms, compression algorithms, etc.

Three artificial test benchmarks, listed in Table 3, are used to stress test WCET analysis tools.

The two parallel benchmarks, listed in Table 4, are: Debie and PapaBench. These two benchmarks are comparable in size and in the number of tasks.

The application benchmarks, derived from real applications and provided with simulated input stimuli, are listed in Table 5. Lift is a lift controller that has been deployed at a factory in Turkey. The hardware is based on a Java processor (JOP). The controller has just a few inputs (command buttons and input sensors for the height measurement) and a simple motor

Table 3 TACLeBench test benchmarks.

Name	Description	Code Size (SLOC)	Origin
cover	Artificial code with lots of different control flow paths	620	MRTC
duff	Duff's device	35	MRTC
test3	Artificial WCET analysis stress test	4235	Universität des Saarlandes

Table 4 TACLeBench parallel benchmarks.

Name	Description	#Tasks	Code Size (SLOC)	Origin
Debie	DEBIE-1 instrument observing micro-meteoroids and small space debris	8	6615	Tidorum Ltd
PapaBench	UAV autopilot and fly-by-wire software	10	6336	Paparazzi

control. The I/O devices are simulated in the benchmark. The Java version of Lift is part of the Java benchmark suit JemBench [13]. Benedikt Huber has translated lift to C.

powerwindow implements a controller for an electric window in a car. Both the driver and the passenger are able to control the window by requesting the window to roll up or down. In case an object is stuck between the window and the doorframe, the controller will move the window down to avoid damaging the object.

3.2 Issues with the Original Sources

The original benchmarks include all input data or we added input data into the C source. However, this effectively turns them into single-path programs. This fact could be used by analysis tools to explore only this single path. Another consequence of the fixed input data, and that some programs do not provide any return value, is that compilers with optimizations turned on can optimize most of the code away. However, to prohibit the unwanted compiler optimizations we changed the way input data is represented in variables (made them `volatile`), and made the return of `main` dependent on the benchmark calculation.

Some benchmarks contain target dependent code. For example, PapaBench contains hardcoded I/O addresses. Furthermore, a few benchmarks (e.g., rijndael) are byte order dependent and there is no standard way in C to detect the byte order of a processor. Finally, some benchmarks can be executed only once, either because they rely on global initialization, or because they use `malloc` but not `free`.

3.3 Benchmark Updates

The benchmarks have been rewritten to split the functions of input data initialization, the benchmark itself, and computing a return value depending on the output data. Moving the input data generation into its own function resulted sometimes in movements from originally stack allocated data into global data. All function and variable names are prepended with

Table 5 TACLeBench application benchmarks.

Name	Description	#Tasks	Code Size (SLOC)	Origin
lift	A lift controller	1	361	Martin Schoeberl
powerwindow	Distributed power window control	4	2533	CoSys-Lab

the benchmark name to provide unique names. All loops have been annotated with loop bounds. Moreover, several bugs have been fixed, and compiler warnings have been eliminated. The benchmarks are now ISO C99 compliant. Some benchmarks have been renamed. The original name of a benchmark can be found in the comment header of each benchmark. The library functions used by some benchmarks have been moved to their own files. All source files adhere to a common set of formatting rules that can be found in the git repository (*doc/code_formatting.txt*).

Due to these changes, results obtained with the TACLeBench versions of these benchmarks are not comparable with the original versions of the benchmarks.

3.4 Licenses

An issue we encountered with several benchmarks was that the original source did not include any licensing information. In absence of such information, we had to assume that the copyright holder reserves all rights. Wherever necessary, we contacted the copyright holders to obtain the right to use, modify, and redistribute the benchmarks. In a small number of cases we however discovered that the code was in fact under a license that made the benchmark unusable; the respective benchmarks were consequently dropped from the TACLeBench benchmark suite. All benchmarks in the benchmark suite now contain licensing information, such that future developments do not require tracking down the original authors of the benchmark.

3.5 Usage Recommendations

TACLeBench is released in source form. However, to report results based on TACLeBench, the benchmarks shall not be changed. Furthermore, the version of TACLeBench shall be included in any paper.

If possible, use all benchmarks in your evaluation. One issue with subsetting a benchmark collection is to *selecting the most representative benchmarks* to show an improvement. However, this introduces a bias in the result and is considered scientific misconduct. If you really need to subset a benchmark collection you have two possibilities: (1) use only one class of benchmarks, e.g., the sequential benchmarks or (2) use a random selection, possibly generated by a program.

If some benchmarks have not been used, state the reason for exclusion (e.g., "the tool/target architecture does not support floating point numbers").

3.6 Known Uses of TACLeBench

Although TACLeBench is a relative new collection of embedded benchmarks, we can already list some usage of the collection in research projects. This early adaption of TACLeBench is already a strong indication of the need of such a benchmark collection.

- Compiler optimizations: The origin of TACLeBench is the collection of free programs used for evaluation of the wcc compiler [4].
- Measurement-based analysis: TACLeBench has been used to evaluate the continuous measurement-based WCET estimation approach presented in [1]. The different characteristics of the different benchmarks proved to be very useful to trigger edge cases in the analysis and led to various improvements of the prototype. However, the evaluation for this approach happened before version 2.0 of TACLeBench has been finished, which makes the results non-comparable to the final benchmark collection.
- Hybrid analysis: The hybrid model splits the code of tasks into basic blocks and uses measurements to obtain instruction traces. The challenge of this two-layer hybrid approach is tackling the computational complexity problems within the static analysis and accuracy within the measurements based layer. The TACLeBench is used in the COBRA-HPA (COde Behaviour fRAmework-Hybrid Program Analyser) framework that facilitates evaluation of the different approaches using different block sizes. Furthermore, COBRA-TG (Taskset Generator) uses TACLeBench for schedulability analysis. Different scheduling methodologies can be analyzed in a reproducible way using generated tasksets based on specific application descriptions.
- Hardware design: As the TACLeBench collection is self contained, it leads itself as an easy to use benchmark collection to evaluate computer architecture design in the embedded and real-time domain. We have used version 1.0 for the evaluation of the stack cache design optimized for real-time systems [12].

3.7 Source Access and Compiling the Benchmarks

The benchmarks are hosted on GitHub at `https://github.com/tacle/tacle-bench`. Each benchmark is in its own folder and can simply be compiled with your favorite C compiler with following command:

```
cc/gcc/clang *.c
```

4 Evaluation and Sanity Checks

To evaluate the complexity of the benchmarks and their resilience against compiler optimizations, we executed each benchmark on `pasim`, a cycle-accurate simulator of the Patmos architecture [11]. Each benchmark was compiled using `patmos-clang` with activated compiler optimizations (i.e., -O2). The results of this evaluation are summarized in Figure 1. All execution traces start at the function marked as entry point by a pragma directive in the source code of the benchmark. The execution times range from 305 cycles (binarysearch) up to 1,658,333,567 cycles (test3). To put this into relation, the benchmark program test3 runs approximately for 21 seconds on the Patmos platform assuming a CPU Frequency of 80 Mhz. From this evaluation we make the main observation that TACLeBench consists of both short- and long running benchmarks with a huge variety in execution time. No benchmark in the suite is optimized to a single return statement.

Different members of the TACLe COST action changed the code. Such a collaborative procedure is inherently error-prone. Human errors can occur and the original sources were often faulty to start with. The distributive work on the benchmarks led to an additional source of error detection: a benchmark behaving well under system configuration A may be faulty under configuration B. To ensure the quality of the benchmarks and to improve portability, we have thus implemented automatic sanity checks.

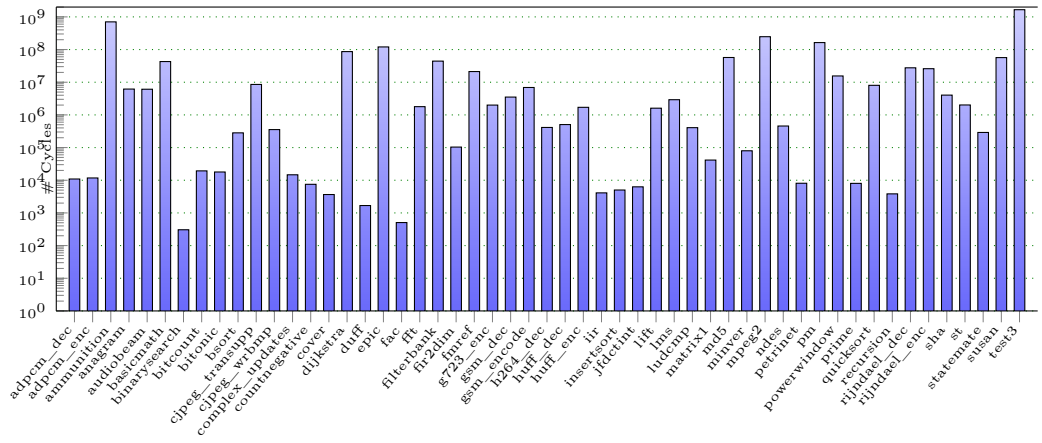


Figure 1 Execution times of programs in TACLeBench range from 305 cycles up to more than 1,600,000,000 cycles on the Patmos architecture.

The code quality is validated centrally and the latest results can be viewed online.[4] For this sanity check, all benchmarks are compiled using *gcc*, *g++* and *clang*, executed and the return values are checked against the expected value. Clang's static analyzer[5] is used to further reveal programming bugs, such as out-of-bounds errors, and also to validate the compliance of the code with the code-formatting rules.

The portability has been checked via a shell-script (`checkBenchmark.sh`) that is now part of the TACLeBench repository. The script allows to quickly identifying incompatibilities of the benchmarks with specific operating systems, compilers, and system configurations. Even though full coverage of all system configurations can never be achieved, we were able to cover most of the common operating systems and compilers.

5 Conclusion

Research in the field of embedded real-time systems needs benchmarks to evaluate research ideas. TACLeBench provides an open source collection of 53 benchmarks. As all benchmarks are self-contained, they are easy to use in systems that are lacking the standard library or an operating systems. 2.0 will be released when the last missing programs have been adapted to the common coding style. We intend to grow the collection of programs and coordinate the release of benchmark versions. We further welcome contributions to the benchmark collection.

Acknowledgements. We want to thank Bendikt Huber for porting the Lift benchmark from Java to C. We want to thank Niklas Holsti from Tidorum Ltd for contributing DEBIE in open-source. We want to thank Yorick De Bock, Florian Kluge, Jörg Mische, and Haoxuan Li for helping in restructuring the benchmarks.

References

1 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In

[4] `https://www4.cs.fau.de/Research/TACLeBench`
[5] `http://clang-analyzer.llvm.org/`

16th Int'l Workshop on Worst-Case Execution Time Analysis (WCET 2016), volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2016.4.

2 EDN Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

3 Heiko Falk, Timon Kelter, Robert Pyka, and Daniel Schulte. TACLeBench Flow Facts Documentation. White paper, September 2013.

4 Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-Aware C Compiler. In *6th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2006.673.

5 Jan Gustafsson. The worst case execution time tool challenge 2006. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 233–240, Paphos, Cyprus, November 2006. IEEE. doi:10.1109/ISoLA.2006.72.

6 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 136–146, 2010. doi:10.4230/OASIcs.WCET.2010.136.

7 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 *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, 2001.

8 Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, St. Louis, MO, United States, April 2008. IEEE Computer Society. URL: http://www.jopdesign.com/doc/volta_rtas2008.pdf, doi:10.1109/RTAS.2008.34.

9 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. *European Space Agency – Publications*, 457:307–312, 2000.

10 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, 2006. URL: 10.4230/OASIcs.WCET.2006.678.

11 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. URL: http://www.jopdesign.com/doc/t-crest-jnl.pdf, doi:10.1016/j.sysarc.2015.04.002.

12 Martin Schoeberl and Carsten Nielsen. A stack cache for real-time systems. In *Proceedings of the 18th IEEE Symposium on Real-time Distributed Computing (ISORC 2016)*, York, United Kingdom, May 2016. IEEE. URL: http://www.jopdesign.com/doc/abc.pdf.

13 Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM. URL: http://www.jopdesign.com/doc/jembench.pdf, doi:10.1145/1850771.1850789.

Expressing and Exploiting Conflicts over Paths in WCET Analysis*

Vincent Mussot[1], Jordy Ruiz[2], Pascal Sotin[3], Marianne de Michiel[4], and Hugues Cassé[5]

1 **IRIT, University of Toulouse, Toulouse, France**
`mussot@irit.fr`
2 **IRIT, University of Toulouse, Toulouse, France**
`ruiz@irit.fr`
3 **IRIT, University of Toulouse, Toulouse, France**
`sotin@irit.fr`
4 **IRIT, University of Toulouse, Toulouse, France**
`Marianne.De-Michiel@irit.fr`
5 **IRIT, University of Toulouse, Toulouse, France**
`casse@irit.fr`

Abstract

The presence of infeasible paths in a program is a source of imprecision in the Worst-Case Execution Time (WCET) analysis. Detecting, expressing and exploiting such paths can improve the WCET estimation or, at least, improve the confidence we have in estimation precision. In this article, we propose an extension of the FFX format to express conflicts over paths and we detail two ways of enhancing the WCET analyses with that information. We demonstrate and compare these techniques on the Mälardalen benchmark suite and on C code generated from Esterel.

1998 ACM Subject Classification B.2.2 Performance Analysis and Design Aids, C.3 Special-Purpose and Application-Based Systems, C.4 Performance of Systems, D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, Infeasible paths, Path conflicts, IPET, CFG transformation

Digital Object Identifier 10.4230/OASIcs.WCET.2016.3

1 Introduction

The Worst-Case Execution Time (WCET) analysis of a program takes into account all the finite (with loop bounds) paths of its Control Flow Graph (CFG). The data manipulated by the program might make some of these paths infeasible. If the WCET analyser is unaware of these infeasible paths, or is not able to exploit them, the analysis *may* suffer from:

1. Direct over-approximation when the Worst-Case Execution Path (WCEP) is infeasible,
2. Indirect over-approximation when infeasible paths pollute the timing of hardware analyses.

The function shown on Listing 1 illustrates these two points. When called with a value less than 64 the heavy computation `comp` is infeasible and should not be taken as WCEP (Item 1). When called with a value greater than or equal to 64 the array `precomp` is not accessed and should not alter the abstract state of the cache (Item 2).

* This work is supported by the French ANR W-SEPT.

```
int f(int n) {
  if (n < 64) return precomp[n];
  else return comp(n);
}
```

Listing 1 A function that either computes its result or returns a precomputed result.

```
<conflict>
  <!-- Edge or block identifier 1 -->
  <!-- ... -->
  <!-- Edge or block identifier N -->
</conflict>
```

Listing 2 General form of a conflict.

In this paper we focus on a specific class of infeasible paths called *conflicts*[1]. A conflict is defined by a set of CFG edges that cannot all appear in the same execution trace. As soon as one of these edges can be executed several times, there is no straightforward translation of the conflict into the Integer Linear Program (ILP) usually built to compute an upper bound on the WCET – Implicit Path Enumeration Technique (IPET). If ILP could handle efficiently disjunctions then we could translate a conflict between the edges `A`, `B` and `C` by $n_{\texttt{A}} = 0 \vee n_{\texttt{B}} = 0 \vee n_{\texttt{C}} = 0$ where $n_{\texttt{X}}$ is the number of executions of edge `X`.

Our contributions are the following:

- We present an extension of the FFX format [2] for expressing conflicts. We can represent simple conflicts, conflicts that are valid in a given context and conflicts involving specific instances of an edge, with or without relevant order (Section 2).
- We propose two ways of integrating conflicts in the WCET analysis, through CFG transformation (based on [8], Section 3) or with additional ILP constraints (based on [9], Section 4).

In Section 5 we experiment these approaches on the Mälardalen benchmark suite [4] and two C programs generated from Esterel, considering conflicts inferred by a SMT-based tool at binary level [10]. Gains can be significant (but do not exceed 10%).

The issue of infeasible paths in WCET analysis being an active topic since many years, we compare our expression format and integration process with some previous work inferring, expressing and exploiting conflicts [3, 5, 11, 6] in Section 6.

2 Expressing Conflicts in FFX

The open format FFX (Flow Facts in XML) is a flowfact annotation language [2] aimed to be portable, expandable and easy to write, understand and process. Annotations in FFX are stored in an XML file rooted by a `flowfacts` element. Inside, a hierarchy of elements represents:

- Facts (e.g. loop bounds)
- Context of validity for inner elements (e.g. call context, loop iteration)

2.1 The `conflict` Element

We enrich the FFX format with the element `conflict` illustrated in Listing 2. This element means that a valid program trace cannot contain all blocks/edges mentioned inside. In other terms, only paths that go through at most N-1 of the N blocks/edges mentioned in the element are valid. A block/edge is identified by the `block`/`edge` FFX element and we make the assumption that it can be mapped to a block/edge of the CFG.

[1] The term is borrowed from [11].

```
<loop loopId="L">
  <iteration number="*">
    <conflict>
      <edge "A" />
      <edge "B" />
    </conflict>
  </iteration>
</loop>
```

Listing 3 Conflicts for each iteration of a loop.

```
<conflict>
  <edge "A" />
  <call name="C1" ...>
    <edge "B" />
    <edge "C" />
  </call>
</conflict>
```

Listing 4 Conflict with edges in a call.

```
<conflict ordered="yes">
  <edge "A" />
  <edge "B" />
  <edge "C" />
</conflict>
```

Listing 5 `A` then `B` then `C` is infeasible, but `CAB` is allowed for example.

2.1.1 Context of Validity

We inherit from FFX the notion of context of validity. A `conflict` element inside a given context applies to each sub-trace defined by the context. For example, Listing 3 describes a conflict that occurs in each iteration of loop `L`. It means that in one iteration we can see no `A` but some `B` and in another iteration no `B`, but some `A` (no `A` and no `B` is also allowed).

2.1.2 Specific Instances of Edge/Block

The use of contextual elements in FFX allows stating that a property holds in that context (e.g. for the last iteration, when the function is called from a given call site). We employ the same contextual elements *inside* the `conflict` tag to restrain it to specific instances of a given edge or block. For example, Listing 4 describes a conflict between `A` and instances of `B` and `C` belonging to a specific call.

2.1.3 Ordered Conflict

We experienced that a conflict as described in Section 2.1 is a strong property in the sense that it holds regardless of the order of its elements. To allow the expression of a weaker property, we introduced an attribute named `ordered` for the `conflict` element. This attribute can be given the value `yes` or `no`, the default being `no`. If a conflict is ordered, it only states that its constituents cannot appear altogether in that order.

2.2 Formal Semantics

In this section, we give a formal semantics to the `conflict` element. This semantics is the foundation of the properties of Section 2.3 but it can be skipped if reading Section 2.1 was clear enough. The semantics takes the form of a predicate indicating whether the execution path π is accepted or not. We *extend* the FFX acceptance semantics $\mathcal{FFX}[\![\cdot]\!]$ as follows:

$$\mathcal{FFX}\left[\!\!\left[\begin{array}{c}\texttt{<conflict ordered=}o\texttt{>}\\ elems\ \texttt{</conflict>}\end{array}\right]\!\!\right](\pi) = \mathcal{C}[\![elems]\!](o,\pi)$$

$$\mathcal{C}[\![elem_1 \dots elem_n]\!](\texttt{no},\pi) = \bigvee_{1\le i\le n} \mathcal{C}[\![elem_i]\!](\texttt{no},\pi)$$

$$\mathcal{C}[\![elem_1 \dots elem_n]\!](\texttt{yes},\pi) = \langle\sigma_1,\dots,\sigma_n\rangle \in \text{split}_n(\pi) \Rightarrow \bigvee_{1\le i\le n} \mathcal{C}[\![elem_i]\!](\texttt{yes},\sigma_i)$$

$$\mathcal{C}[\![\texttt{<edge } e\texttt{/>}]\!](o,\pi) = e \notin \pi$$

$$\mathcal{C}[\![\texttt{<block } b\texttt{/>}]\!](o,\pi) = b \notin \pi$$

$$\mathcal{C}[\![\texttt{<}ctx\texttt{>}elems\texttt{</}ctx\texttt{>}]\!](o,\pi) = \sigma \in \text{sub}_{ctx}(\pi) \Rightarrow \mathcal{C}[\![elems]\!](o,\sigma)$$

where sub_{ctx} is a function returning the sub-traces matching the contextual element *ctx* and where $\sigma_1.\cdots.\sigma_n = \pi$ means that the σ_i form a sequential decomposition of π.

2.3 Properties of the `conflict` Element

▶ **Property 1.** *A* `conflict` *element can be flipped with:*

- *Its internal context, if it is the only child of the* `conflict`*,*
- *Its external context, if the* `conflict` *is its only child.*

```
<conflict>
  <context>
    <!-- content -->
  </context>
</conflict>
```

$\Longleftrightarrow$

```
<context>
  <conflict>
    <!-- content -->
  </conflict>
</context>
```

▶ **Property 2.** *A conflict having its* `ordered` *attribute set to* `no` *entails the same conflict with this attribute set to* `yes` *since the paths that go through an ordered list of edges is a subset of the paths that go though the same list of edges in any order.*

```
<conflict ordered="no">
  <!-- content -->
</conflict>
```

$\Longrightarrow$

```
<conflict ordered="yes">
  <!-- content -->
</conflict>
```

3 Integrate Conflicts by CFG Transformation

In this section we propose to turn a `conflict` FFX tag into an equivalent automaton to integrate it in the analysis process through a CFG transformation. This idea is based on [8] where we present a method to turn the semantic information of an annotation language (such as FFX) into a hierarchical automaton enriched with constraints. We then perform a product operation between the CFG of a program and this automaton and feed the result to IPET to obtain a WCET.

3.1 Example of CFG Transformation

If we reduce a CFG to an equivalent Deterministic Finite Automaton (DFA), we obtain the set of paths that are structurally possible in a program. If we want to remove an edge of this DFA, we can simply perform an automata product with an automaton that would forbid that specific edge. More precisely, this automaton would accept any edge except this one.

The general idea is to rely on the formal operations that exist on automata to integrate additional semantic information in the analysis process.

Figure 1 illustrates an automata product between an automaton that represents a conflict between two edges A and B and the DFA that corresponds to a CFG.

The *Conflict* automaton works as follows: in the state `x`, all edges are accepted through the default transition *, except edges A and B that go respectively in `y` (resp. `z`) where any edges different from B (resp. A) is accepted.

The result of this product is an unfolded CFG where there exists no path that can go through A and B. The semantic information of the conflict has been carried by the automaton and integrated as a structural restriction in the CFG.

Two limitations appear in this process:

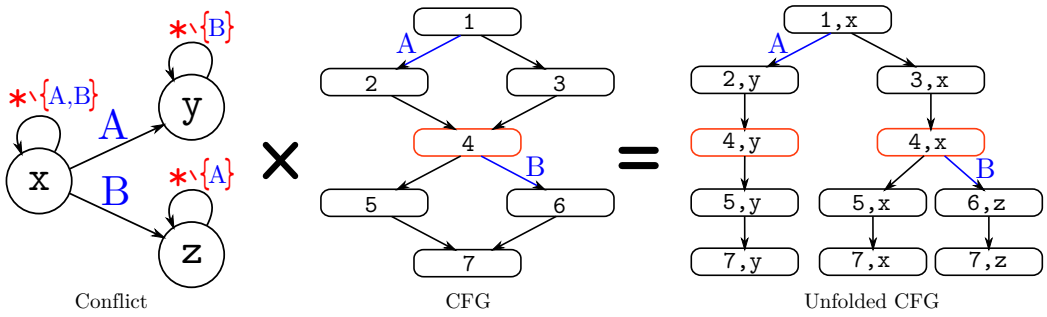


Figure 1 Product between a conflict automaton and a CFG.

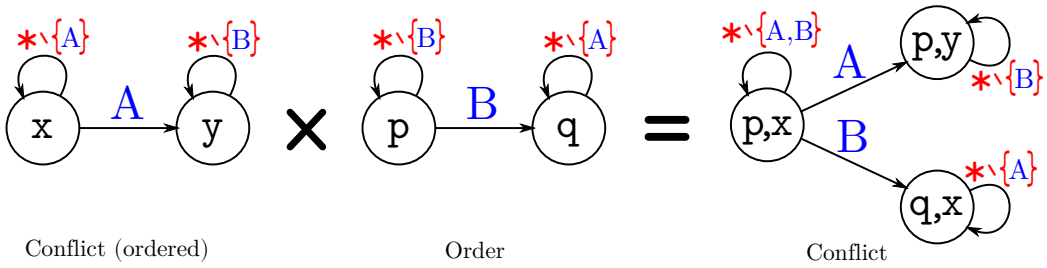


Figure 2 Product between an ordered conflict and the automaton that describes the order.

- Turning a semantic property into an equivalent automaton is not straightforward. However we presented in [8] a strong formalism of hierarchical automata enriched with constraints that can handle most of the annotations that are used in the WCET analysis (e.g. contexts, loop bounds, conflicts...).
- This approach suffers from scalability problems: in Figure 1, some blocks are duplicated due to the product (e.g. block 4,5,7). Also, blocks `7,x` and `7,y` are now separated due to the `z` state, even if there exists no path that could go through A after B. And if the CFG had more blocks after block 7, they would all appear in the three branches. Moreover this is only a 2-edge conflict. A 3-edge conflict has seven states and a product could almost multiply the number of blocks of a CFG by seven. We propose to use the "ordered" property to reduce the number of states of the conflict automaton and then to reduce the number of replications.

3.2 Simpler Automaton with the Ordered Property

When the ordered property of a conflict is set to "yes", the only paths removed are those that go through the elements of a `conflict` in the right order. This is weaker than the non-ordered conflict. However, if we add the information that the order specified in the `conflict` is the only one possible, we obtain an equivalence to the non-ordered conflict.

The first automaton in Figure 2 represents an ordered conflict that only excludes paths where B is taken after a A.

If we perform a product with the second automaton which ensures that no A can occur after a B, we obtain the previous non-ordered conflict.

It turns out that when our conflict detection tool finds a conflict over a list of edges of a CFG in a specific context, it also ensures that there exists no other possible orders for these edges in this context. In other terms, the order of these edges is already a structural constraint of the CFG. In these conditions, we can carry the property with the weaker but simpler ordered conflict automaton, since it is the only structurally feasible path in the CFG.

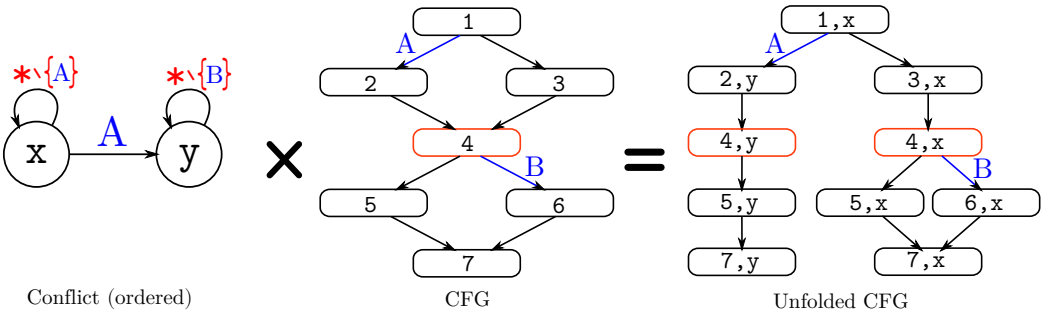


Figure 3 Product between an ordered conflict and a CFG.

3.3 Benefits of the Ordered Property

Figure 3 illustrates how a simpler conflict automaton avoids the separation of the CFG in three distinct branches while still removing the conflicting path from the structure.

With this ordered version of the conflict, we can consider performing the product using a conflict over n edges, even when n is greater than three since it results in an automaton with $n-1$ states. It was impossible with the non-ordered conflict which results in an automaton with $2^n - 1$ states and almost as many potential replications of each block of the CFG.

4 Integrate Conflicts with ILP Constraints

In this section, we present a method based on [9] for translating FFX conflicts into ILP constraints. These constraints are meant to be added to the constraints produced by an IPET-based WCET analysis tool (with potential pessimism).

4.1 Framework

In [9] Raymond proposed a method for turning any set of conflicting edges into linear constraints. For example, consider a CFG where an edge $\mathtt{A}$ is located inside a loop (bound n) and an edge $\mathtt{B}$ outside. Consider a conflict between the edge $\mathtt{A}$ (in any iteration) and the edge $\mathtt{B}$. If we unroll the loop, the complete set of conflicts is $\{(\mathtt{A}_1, \mathtt{B}), \ldots, (\mathtt{A}_n, \mathtt{B})\}$. Note that $\mathtt{A}_1, \ldots, \mathtt{A}_n$ are called *avatars* of $\mathtt{A}$.

The method proposed turns this set of conflicts into the ILP constraint $\mathtt{A} + n.\mathtt{B} \leq n$.

The key idea of this framework is that it works on an acyclic unfolding of the original CFG where we can turn a conflict like $(\mathtt{A}_1, \mathtt{B})$ into $\mathtt{A}_1 + \mathtt{B} \leq 1$. From sets of such inequalities, clever summation recovers linear constraints on the original CFG. This leads to a general formula that is valid for any set of CFG edges X and any set of conflicting edge avatars S built upon X:

$$\sum_{x \in X} p_x x \leq (|X| - 1)|S| + \sum_{x \in X} l_x$$

where:

- p_x is the maximum of the counts of each avatar of x in S (called multiplicity),
- l_x is $p_x m_x - |S|$ (called lack),
- m_x is the number of avatars of x in the unfolding.

This formula does not reflect exactly the conflict, but it is a safe approximation.

```
<conflict ordered="yes">
  <edge "a" />
  <edge "b" />
</conflict>
```

Listing 6 Conflict without context.

```
<conflict ordered="yes">
  <loop address="0x...">
    <iteration number="*">
      <edge "a" />
      <edge "b" />
    </iteration>
  </loop>
</conflict>
```

Listing 7 Conflict on each iteration of a loop.

```
<conflict ordered="yes">
  <loop address="0x...">
    <iteration number="-1">
      <edge "a" />
      <edge "b" />
    </iteration>
  </loop>
  <edge "c" />
</conflict>
```

Listing 8 Conflict on the last iteration of a loop and after.

4.2 Translations

Our contribution is a prototype Otawa [1] plug-in to translate FFX conflicts into ILP constraints. We illustrate the translation performed on several examples.

The conflict presented in Listing 6 is a simple conflict with no contexts. Under the hypothesis that $m_a = m_b = 1$ (a and b appear only once in the acyclic unfolding) the method results in: $S = \{(a, b)\}$ and $|S| = 1, p_a = p_b = 1, l_a = l_b = 0$. Therefore the generated constraint is $a + b \leq 1$.

The Listing 7 presents a conflict for each iteration of a loop. Under the assumption that $m_a = m_b = n$, we obtain the system:

$$\begin{cases} S = \{(a_1, b_1), \ldots, (a_n, b_n)\} \\ |S| = n, p_a = p_b = 1, l_a = l_b = 0 \end{cases}$$

The resulting ILP constraint is then $a + b \leq n$. Note that the right-side of the inequality can be generalized to $n(|X| - 1)$.

Finally, the conflict presented in Listing 8 illustrates a conflict between two edges in the last iteration of a loop and a third one after. Under the assumption that $m_a = m_b = n \wedge m_c = 1$, the following system is obtained:

$$\begin{cases} S = \{(a_n, b_n, c)\} \\ |S| = 1, p_a = p_b = p_c = 1, l_a = l_b = n - 1, l_c = 0 \end{cases}$$

The generated ILP constraint is then $a + b + c \leq 2n$. Again, note that the right-side of the inequality can be generalized to $|X| + |In|.(n - 1) - 1$ where $In \subseteq X$ is the subset of edges belonging the loop.

5 Experiments

The PathFinder tool presented in [10] runs an abstract interpretation top-to-bottom analysis on binary programs, looking for semantic conflicts. It acts as a pre-analysis for the WCET analysis, aiming to provide information about infeasible paths in the most factorized, exploitable way possible for other analyses. It relies on the OTAWA framework.

This infeasible path analysis tool models the program state for a set of paths in the CFG as a conjunction of predicates on registers and memory cells. It checks the feasibility of a set of paths by checking the satisfiability of this abstract program state in an SMT[2] solver.

When an infeasible path is detected, PathFinder attempts to express the detected conflicts in a *minimal* number of infeasible paths, each written as a list of CFG edges that cannot all

[2] Satisfiability Modulo Theory

Table 1 Results on the Mälardalen benchmarks.

	Nb. of conflicts found		WCET gain simple arch.		WCET gain arm9 + cache	
Program	Total	After minim.	Constraints	Unfolded	Constraints	Unfolded
Small Mälardalen benchmarks						
adpcm	174	28	0.00 %	0.00 %	CE	CE
cnt	118	5	0.00 %	0.00 %	0.00 %	0.00 %
cover	3	3	6.95 %	6.95 %	**0.01 %**	**0.25 %**
crc	8	8	0.50 %	0.50 %	**4.10 %**	**9.70 %**
edn	7	6	0.03 %	0.03 %	CE	CE
expint	8	5	0.00 %	0.00 %	**0.00 %**	**0.09 %**
fibcall	1	1	0.72 %	0.72 %	0.32 %	0.32 %
fir	1	1	0.00 %	0.00 %	**3.37 %**	**7.45 %**
select	18	11	0.16 %	0.16 %	0.09 %	0.09 %
sqrt	407	10	0.40 %	0.40 %	0.04 %	0.04 %
Large Mälardalen benchmarks						
statemate	1118	71	2.77 %	CE*	1.00 %	CE*
ud	13	1	1.17 %	1.17 %	1.08 %	1.08 %
nsichneu	13648	7684	0.00 %	CE*	0.00 %	CE*
minver	10	9	1.40 %	1.40 %	CE	CE
ludcmp	29	3	0.00 %	0.00 %	0.00 %	0.00 %
lms	2097	141	CE	CE	CE	CE
fft1	830	149	CE	CE	CE	CE
qurt	797	41	CE	CE	CE	CE
Esterel benchmarks						
runner	5618	185	9.84 %	CE*	9.12 %	CE*
abcd	4949	274	3.01 %	CE*	5.17 %	CE*

be taken on a single path associated with a context (the scope of a loop, of a function for any or a particular function call point).

This list of edges expressing each infeasible path must also be as small as possible, in order to minimize the complexity of WCET analysis that will use these results. This is achieved by retrieving from the SMT solver a kernel of predicates, named an unsatisfiable core, and hook one or several edges to each predicate. Other refinements are performed, namely using (post-)dominance properties on the CFG in order to remove superfluous items from the list of conflicting edges.

Once the analysis completes and the infeasible paths have been minimized as best we can, we output them using the FFX format (Section 2).

Results of our experiments are presented in Table 1. Programs were compiled for the *armv5t* architecture without any optimization (-O0). The second and third columns show the total number of conflicts found by Pathfinder, and the number after minimization. In the next two columns, we analyzed the benchmarks with a trivial architecture that applies a simple metric to compute low-level timings and without any cache. In the last two columns, we used an architecture model derived from ARM9 with a 1 KB instruction cache and a 256 KB data cache.

For several programs we were not able to compute a WCET due to *Capacity Exceeded* (CE) because of scalability problems occurring either during the unfolding of the CFG (CE*), or during the WCET computation itself. Other benchmarks of the suite are not listed here either because they contain recursion that is not supported by Pathfinder or because Pathfinder did not detect any conflict.

In the *Constraints* columns we fed ILP with the constraints derived from the conflicts (Section 4) while we applied the automata product (section 3) in the *Unfolded* columns. The values are the percentage of gain between the WCET computed with the integration of conflicts over the WCET estimation without.

Thanks to the integration of conflicts in the WCET analysis, we observed significant improvements of the precision in some cases, even for a trivial architecture without cache: for these benchmarks, the original WCEP was infeasible.

Unsurprisingly, the table also shows that for a trivial architecture without cache, the WCET precision improvement is exactly the same if we add ILP constraints or if we unfold the CFG. In other terms, gains only appear when the WCEP is infeasible.

On the other hand we highlighted an important result in the column where cache is used: significant precision improvements are noticeable (in bold) when we unfold the CFG. Indeed, at some points in the WCET analysis, abstract cache states that represent the possible states of a cache are merged. When the constraint method is used, the merge points remain unchanged, but unfolding the CFG allows to separate paths which avoids some of these merging operations. It results in more precise abstract cache states and eventually a more precise WCET estimation.

6 Related Work

Engblom et al. [3] present an IPET-based framework for WCET analysis together with an annotation language based on scopes. These scopes correspond to the contexts enclosing our conflicts. We rely on their technique for scoping out local ILP constraints to global ones.

Kirner et al. [6] survey the annotation languages used by the WCET analysis tools (including FFX). They identify categories of dynamic control flow information but the notion of conflict is absent. In their section 9, they use four languages to encode flow information, one of which ($\mathbf{B}_1$) is a contextual conflict. Only PL and IDL are able to encode faithfully the conflict using a regular expression (linked with Section 3).

Suhendra et al. [11] present an algorithm[3] for inferring pairwise-conflicts in a given context (function or iteration). They then use these conflicts to improve a path-based WCET analysis. There tool was not meant to export this knowledge.

Knoop et al. [7] refine the WCET by disproving the feasibility of the WCEP. This approach has the clear interest of focusing the flow analysis on relevant paths. The price to pay is an increased WCET resolution technique complexity. Note that ignoring the infeasibility of paths that are not WCEP might lead to indirect over-approximation of the WCET (Item 2 in Section 1).

Ruiz and Cassé [10] retrieve conflicts from a binary program. Unlike the work previously mentioned these conflicts can involve edges in a loop and edges outside of this loop. The analysis can deliver contextual conflicts over a single function call. It was a motivation for extending the FFX format so to encompass all these subtleties.

[3] It turns out that Algorithm 1 of [11] is wrong. If the program of Figure 1 is run with $\mathtt{x} = \mathtt{y} = \mathtt{z} = 5$ the path taken goes through edges that are reported as branch-branch conflict. This is due to a bad hypothesis in the second item of Definition 3.2: *all* paths between the two branches must not modify the involved variables.

7 Conclusion

Throughout the paper, we presented an extension to the FFX format that allows representing a class of infeasible paths called conflicts (Section 2). Finding these conflicts and taking them into account enables an improvement of the WCET analysis precision of up to 10% on some programs (Section 5). We presented two methods to integrate the conflicts in the WCET analysis. One creates additional ILP constraints reflecting the conflicts (Section 4) and scales well. The other one is an automatic transformation of the CFG that removes the conflicting paths (Section 3); the size of the CFG may explode, but this method can affect the low-level analyses and yield additional WCET precision.

We noted that benchmark suite programs offer a field of improvement through detection and integration of conflicts, while source codes generated from Esterel compiler are slightly more promising. We plan to look for other classes of programs to illustrate the various conflicts that our tools are able to detect and take into account. We also plan to address the issue of scalability, both in the WCET analysis and with the unfolding method, in order to support bigger and more complex programs.

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*, 2010. doi:10.1007/978-3-642-16256-5_6.

2 Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX: a portable WCET annotation language. In *20th International Conference on Real-Time and Network Systems (RTNS 2012)*, 2012. doi:10.1145/2392987.2392999.

3 Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *21st IEEE Real-Time Systems Symposium (RTSS 2000)*, 2000. doi:10.1109/REAL.2000.896006.

4 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, 2010. doi:10.4230/OASIcs.WCET.2010.136.

5 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *27th IEEE Real-Time Systems Symposium (RTSS 2006)*, 2006. doi:10.1109/RTSS.2006.12.

6 Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 10(3), 2011. doi:10.1007/s10270-010-0161-0.

7 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: on-demand feasibility refinement for proven precise wcet-bounds. In *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, 2013. doi:10.1145/2516821.2516847.

8 Vincent Mussot and Pascal Sotin. Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2015)*, 2015. doi:10.1109/RTCSA.2015.11.

9 Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *International Conference on Embedded Software (EMSOFT 2014)*, 2014. doi:10.1145/2656045.2656046.

10 Jordy Ruiz and Hugues Cassé. Using SMT solving for the lookup of infeasible paths in binary programs. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015. doi:10.4230/OASIcs.WCET.2015.95.

11 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *43rd Design Automation Conference (DAC 2006)*, 2006. doi:10.1145/1146909.1147002.

Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs*

Boris Dreyer[1], Christian Hochberger[2], Alexander Lange[3], Simon Wegener[4], and Alexander Weiss[5]

1 **Fachgebiet Rechnersysteme, Technische Universität Darmstadt, Darmstadt, Germany**
dreyer@rs.tu-darmstadt.de
2 **Fachgebiet Rechnersysteme, Technische Universität Darmstadt, Darmstadt, Germany**
hochberger@rs.tu-darmstadt.de
3 **Accemic GmbH & Co. KG, Kiefersfelden, Germany**
alange@accemic.com
4 **AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany**
wegener@absint.com
5 **Accemic GmbH & Co. KG, Kiefersfelden, Germany**
aweiss@accemic.com

Abstract

Traditionally, the Worst-Case Execution Time (WCET) of Embedded Software has been estimated using analytical approaches. This is effective, if good models of the processor/System-on-Chip (SoC) architecture exist. Unfortunately, modern high performance SoCs often contain unpredictable and/or undocumented components that influence the timing behaviour. Thus, analytical results for such processors are unrealistically pessimistic. One possible alternative approach seems to be hybrid WCET analysis, where measurement data together with an analytical approach is used to estimate worst-case behaviour. Previously, we demonstrated how continuous evaluation of basic block trace data can be used to produce detailed statistics of basic blocks in embedded software. In the meantime it has become clear that the trace data provided by modern SoCs delivers a different type of information. In this contribution, we show that even under realistic conditions, a meaningful analysis can be conducted with the trace data.

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

Keywords and phrases Hybrid Worst-Case Execution Time (WCET) Estimation for Multicore Processors, Real-time Systems

Digital Object Identifier 10.4230/OASIcs.WCET.2016.4

1 Introduction

In previous work [8], we showed a novel approach for hybrid execution time estimation. Its main features are the precision that we achieve by taking typical cache behaviour into account, the continuous nature of the FPGA-based online aggregation and the non-intrusiveness, as we exploit the hardware tracing mechanisms of modern state-of-the art SoCs.

* This work was funded within the project CONIRAS by the German Federal Ministry for Education and Research with the funding ID 01IS13029. The responsibility for the content remains with the authors.

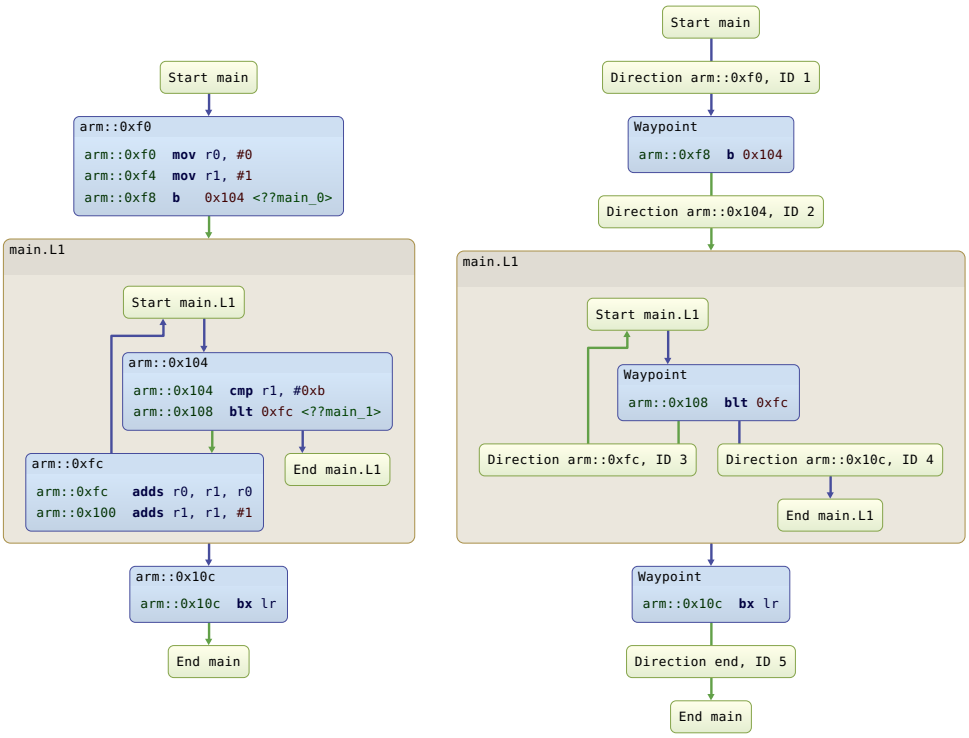


Figure 1 Control flow graph of a small program containing a simple loop (left) and its associated waypoint graph (right).

One of the underlying techniques used to implement this approach is the notion of the control flow graph. A control flow graph consists of basic blocks – sequences of instructions, where each instruction except the first and the last has exactly one predecessor and one successor – and edges that describe the flow of control in a program, i.e. conditionals, routine calls, loops etc. However, the embedded trace unit (ETU) of modern ARM processors (like the Xilinx Zynq featuring an ARM Cortex-A9) is not fully compatible with this model.

The mental model of the ETU is as follows: For each non-linear control flow, for example interrupts and hardware exceptions, but also normal branches and calls, a so-called waypoint event is emitted. These events carry the address where the control flow change happened and the target of the change. Some instructions (the waypoint instructions) always generate a waypoint event [2]. Amongst others, all instructions that possibly modify the program counter are waypoint instructions. This is enough to fully reconstruct the control flow, but less fine grained than the control flow graph.

Consider Figure 1. It contains a control flow graph on the left and its associated waypoint graph on the right. On the left, inside the loop `main.L1`, two basic blocks are shown. The second one, starting at address `0xfc`, does not contain any change-of-flow instruction, but performs always a fall-through to the basic block at `0x104`. Consequently, no waypoint instruction exists that represents this second basic block, but only one for the first basic block (the instruction `blt` at address `0x108`). Each outgoing edge of a waypoint instruction is annotated with the target address given by its waypoint event. Hence we can distinguish the two possible paths through the loop.

To cope with the changed setting, we had to rework large parts of our approach presented in [8]. We spend higher effort in the preprocessing phase, but we are rewarded by a simplified

implementation of the runtime phase. This paper presents the changes that we made to our former approach in order to achieve precise continuous non-intrusive measurement-based execution time estimation for waypoint graphs.

The paper is structured as follows: First, in Section 2, we discuss related work. We continue with a recapitulation of our method's workflow in Section 3. Then, in Section 4, we discuss different hardware tracing units, the quality of the trace they produce and their usefulness for our approach. Afterwards, in Section 5, we highlight the changes between our revised approach and our original approach. Moreover, we introduce a new tool to determine in which context an instruction sequence is executed, the so-called loop automata. We continue with an evaluation of our approach on the TACLeBench benchmark suite [9] in Section 6. Finally, we conclude our work and discuss future work in Section 7.

2 Related Work

The problem of computing tight bounds of the execution time of a program is an active field of research, with many methods and tools using both static and dynamic analysis approaches [14]. Static analysis methods compute safe upper bounds of the execution time from a mathematical model of the target architecture. Dynamic analysis methods, on the other hand, derive the execution time from measurements performed on real hardware. Hybrid methods, like our approach, combine execution time information extracted from measurements with statically computable information like control flow graphs to improve safety, precision and/or coverage of the result. Probabilistic methods, finally, try to compute statistical models from measurements to compute upper bounds of the execution time.

The most basic version of measurement-based execution time analysis, namely end-to-end measurements, is still in frequent industrial use [12], but its problems are manifold. Not only it is unable to produce safe estimates, as in general not all possible scenarios can be measured, but the results are hard to interpret, too, as they are not related to particular parts of the code but only to the whole program.

To overcome this, more structured approaches have been proposed, e.g. by Betts et al. in [6], which combine the measured execution times of small code snippets to form an overall execution time estimate. Their use of software instrumentation leads to the *probe effect*, i.e. the timing behaviour of the program under observation changes due to the used observation technique. Moreover, their method does not account for typical cache behaviour and may be overly conservative. In a more recent publication [7], they use the non-intrusive tracing mechanisms of state-of-the-art debugging hardware. The main obstacle of their method is the limited size of trace buffers and/or the huge amount of trace data. According to their estimates, around half a terabyte of data would be generated in an hour of testing.

Stattelmann et al. [13] propose the use of context information in order to account for cache effects. Their work shows that the inclusion of context information leads to more precise execution time results. However, their approach is limited to processors with sophisticated trace trigger mechanisms and performs again an offline analysis of the collected data.

Most measurement-based methods suffer from one or the other problem mentioned above. Our approach, in contrast, circumvents these drawbacks:

- We measure the timing of short instruction sequences. This fine grained approach allows to see where time is spent.
- We use non-intrusive hardware tracing mechanisms of state-of-the-art processors to produce timestamps. The probe effect is avoided.
- We process the trace events online. There is no need to store huge amounts of trace data.

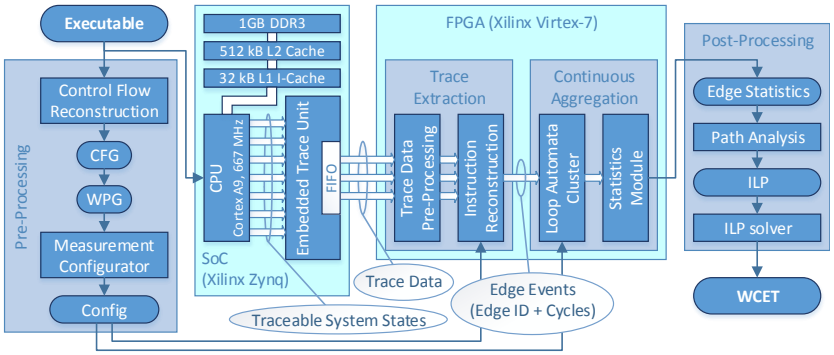


Figure 2 Workflow. Our approach is splitted into three phases: an offline pre-processing phase, the continuous online aggregation phase and an offline post-processing phase.

- We process the trace events continuously. The aggregation can literally run for weeks. The possibilities to catch rare circumstances are increased.
- We incorporate the execution context of instructions and account for typical cache behaviour. The results are thus much more precise.
- The use of an FPGA allows us to adapt our method to different hardware tracing units.

3 Workflow

Our method works on the object code level and is split into three phases: an offline pre-processing phase, the continuous online aggregation phase and an offline post-processing phase. The workflow of our method is shown in Figure 2. The control flow reconstruction and the ILP-based path analysis are re-used from the `aiT` tool chain [1].

We assume that a set of tasks is distributed over the cores of a multicore processor such that each task runs on exactly one core. Each task uses its own trace extraction and continuous aggregation modules. Hence it suffices to describe the workflow for a single core.

Control Flow Reconstruction and Waypoint Graph Computation. First, the binary reader disassembles a fully linked binary executable into its individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. This knowledge is used to form the basic blocks of the control flow graph (CFG).

Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user needs to write some annotations to guide the control flow reconstruction.

After finishing the reconstruction of the CFG, the waypoint graph (WPG) is computed. To do so, a pattern matcher checks for each instruction whether it is a waypoint instruction. Afterwards, the edges of the WPG are computed. For each waypoint instruction found, the algorithm follows the edges in the CFG to find reachable waypoints. This gives the direction of a waypoint edge and its target.

Configuration of the Continuous Online Aggregation. Then, the WPG is used to create a configuration for the trace extraction module as well as for the continuous online aggregation module on the FPGA. This configuration assigns an unique ID to each edge in the waypoint graph. Moreover, it instantiates the lookup tables in the loop automata cluster (see Section 5 for a detailed description).

Trace Extraction. During the program's execution, the ETU continuously emits raw trace data. This stream of data is fed into the trace extraction module. There, the raw data is decoded and compiled into an event stream. An event is generated for each traversal of a waypoint and consists of an ID and a timestamp. The special ID 0 is used if the waypoint does not belong to the WPG computed during the pre-processing phase. This happens for example in case of an interrupt. Otherwise, the ID from the module's configuration is used. The resulting event stream is then fed into the continuous aggregation module.

Continuous Context-Sensitive Aggregation. To achieve precise results, it is important that the aggregation module accounts for cache effects. Typically, the first iteration of a loop needs more time than the subsequent iterations because the instruction cache is not yet filled. Simply aggregating all loop iterations in the same record would thus most probably overestimate the time spend in all iterations but the first. For well-formed loops, we thus compute two statistical records for each edge belonging to a loop, one that aggregates the execution times in the first iteration and another that aggregates the execution times in all subsequent iterations, i.e. we take the execution context into account. This resembles some kind of virtual loop unrolling. If a basic block is part of nested loops, we only discriminate the iterations of the innermost loop, due to limited storage for the statistical records.

The ID of an event is used as input for the loop automata cluster. Each automaton in the cluster performs one step. Then, their state is used to decide whether a loop is executed and if it is the first iteration of the loop or already a later one.

The timestamp of an event is used to measure the execution time of the code snippet represented by the waypoint edge. Various statistics (minimal observed execution time, maximal observed execution time, count of executions) are updated each time an edge event is processed. The ID together with the execution context computed in the loop automata cluster form the index in the memory of statistical records.

Post-Processing and Path Analysis. After the program has finished (or the test engineer has collected enough data), the post-processing phase is started by downloading the statistics from the FPGA's memory. Subsequently, the WPG together with the edge timing statistics are used to construct a maximisation problem encoded as an integer linear program (ILP). Solving this ILP gives a path with maximal execution time (and consequently, an estimate of the worst-case execution time).

Finally, the computed path is visualised for the user. An edge is marked infeasible if no statistics have been created for it. This information can be used to detect dead code. Moreover, the WCET contribution of the individual parts of the program is visualised. That way, the test engineer can see where in the program the hot spots are. This is particularly useful if the program is the target of performance optimisations.

4 Embedded Trace Units

The measurability of the execution time of instruction or basic blocks is a precondition for the proposed hybrid WCET measurement approach.

The traditional software instrumentation methodology with its obvious change of the systems behaviour (application blow up, execution slow down) is inappropriate for this measurement. In lieu thereof ETUs are implemented in the SoC. An ETU observes the SoC internal states, compresses and outputs this information via a dedicated high-bandwidth trace port. There are several ETU implementations available, which differ in the type

Table 1 ETU overview and its applicability for hybrid WCET measurement.

ETU type	Nexus 5001™ [11]		ARM CoreSight™			
Implementation	Traditional branch messages	Branch history messages	ETMv3 [3]		ETMv4 [5]	PFT [2]
Program Flow Observation Level	Branch	Branch	Instruction		Branch	Branch
Cycle count	Yes	No	No	Yes	Yes	Yes
Applicable for hybrid WCET measurement	Yes	No	No	Yes	Yes	Yes

of trace information and the compression efficiency (see Table 1). The most important ETU implementations are Nexus 5001™ [11] (for instance within the NXP Qorriva/QorIQ devices [10]) and the ARM CoreSight™ architecture [4].

The processor can possibly generate more trace data than the SoC's trace port can output at a given time. Therefore, the ETU includes a FIFO to buffer trace messages. The trace processing unit has to be able to handle the overflow of the ETU FIFO if a large volume of trace messages is generated (e.g. at narrow loops with high branch frequency).

5 Revised Method

This section presents a revised version of our approach for hybrid measurement-based timing analysis [8]. The original version of this approach was based on basic blocks, therefore the trace extraction unit had to emit basic block events. These events were also used to determine the execution context of the measured basic blocks, and to compute statistics over these blocks. By considering the execution context of the basic blocks, two statistics per basic block were computed: one containing the execution times of the basic block during the first iteration of its innermost surrounding loop (cold cache) and one containing all subsequent iterations (hot cache).

It turned out that there are processor architectures on the market for which we cannot reconstruct the basic blocks because not enough information is available in the stream of trace data. So we had to revise our architecture to use waypoint edge events instead of basic block events. The module that determined the context of executed instructions based on basic block events had to be replaced by new a one that uses waypoint edge events. This new module is called loop automata cluster and the central point of our revised work. It determines the context of each instruction based on a set of finite state machines and will be further described in this section. Only a few changes had to be made to the original statistics module to be compatible with the new loop automata cluster.

Loop Automata Cluster. The loop automata cluster has the purpose to determine the context of each executed instruction, so that the statistics module can compute context-sensitive statistics. We define the context of an instruction by the context of its innermost surrounding loop. The context of each loop of an application can be determined by interpreting the waypoint edge event stream emitted by the trace extraction module [2]. For this interpretation WPG information is required because the event stream contains only the waypoint ID and the cycle count.

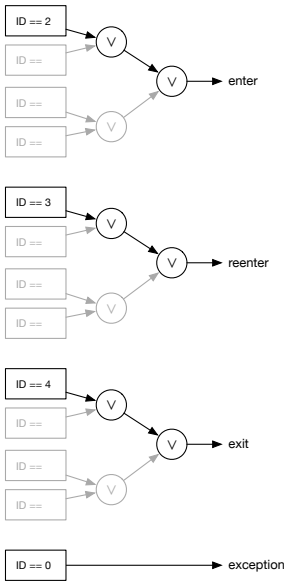


Figure 3 The instantiated comparator trees for the loop in Figure 1.

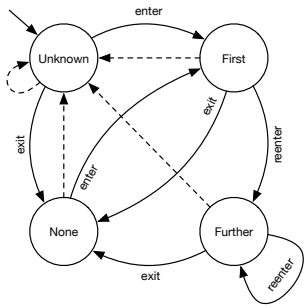


Figure 4 Finite state machine that reflects the different loop contexts.

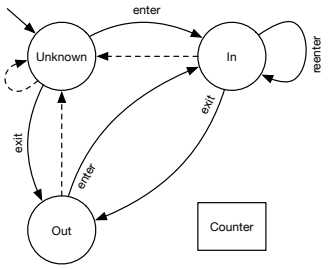


Figure 5 Finite state machine that counts the iterations of a loop.

We model each loop of an application by two finite state machines (FSM) and four comparator trees. Figure 3 illustrates one set of four comparator trees that are used to translate the waypoint edge IDs of the event stream into loop specific context change events, namely `enter` (the loop has been entered), `reenter` (the loop has been iterated), `exit` (the loop has been exited), and `exception` (knowledge about the loop's context have been lost). The compare values of these loop specific comparator tree sets can be extracted from the WPG of the application.

Besides the comparator trees we use FSMs to store loop information. The first FSM gives information about the context of the loop and is illustrated in Figure 4. Its states reflect the different contexts of a loop, namely `None` (the loop is not executed), `First` (the loop is in its first iteration), `Further` (the loop is at least in its second iteration), and `Unknown` (no knowledge whether the loop is executed or not). If the FSM is in state `First`, the statistics for the first iteration of the waypoint are updated. If the FSM is in state `Further`, the statistics for all subsequent iterations are updated. If waypoint edge events have been lost during the trace extraction, e.g. because trace buffers within the processor have been overflowed, it can not be determined whether the loop is executed in the first or further iterations or not. In this case the FSM is in state `Unknown` and both statistics of a waypoint are updated to further maximize its WCET.

During program execution, several loops can be in their first or further iteration, due to nested loops. In this case, the context of the innermost loop determines the context of the waypoint edge events. For this, we use a stack to track the innermost loop during runtime.

The second FSM gives information about the iteration count of the loop and is depicted in Figure 5. It consists of tree states, namely `Out` (the loop is not being executed), `In` (the loop is being executed), and `Unknown` (it is not known if the loop is being executed or not). If the loop is not executed, the FSM is in state `Out` and the iteration counter is zero. Once the loop is executed the state changes to `In` and the counter is set to one because we count the executions of the loop header. Each time the FSM is in state `In` and a `reenter` event

occurs the counter is incremented by one. As soon as the machine changes it state from `In` to `Out` the counter value is considered as performed loop iterations and the loop bounds statistics for this loop are updated.

It is possible that a trace analysis starts after the program execution has been stated. Consequently, there is a lack of loop context information at the beginning of the analysis. Therefore the initial state of each FSM is `Unknown`.

6 Evaluation

We evaluated our approach on a set of benchmarks. However, parts of the prototypical implementation have been simulated in software due to the changes we had to implement compared to our initial approach. We plan to have a full hardware implementation at the time of the workshop.

Setting. The target SoC for our prototype is a Xilinx Zynq featuring a dual-core ARM Cortex-A9 running at 667 MHz. The memory subsystem of this SoC constists of 32 kilobytes of L1 instruction cache, 512 kilobytes of L2 cache and 1 gigabyte of DDR main memory. We deactivated the L2 cache and the dynamic branch prediction in order to focus on L1 instruction cache effects. We used the TACLeBench benchmark collection [9] for the evaluation. We started with the evaluation before version 2.0 of the benchmark collection was finalized and had some problems with some of the tests. In particular, the benchmark `sha` could not be compiled with the C++ compiler provided with the Xilinx SDK 2014.4 that we used. We ran the triplet of a benchmark's `init`, `main` and `return` functions ten times in a row, except for `powerwindow`, which has been run only once as it contains a slightly different structure than the other benchmarks. Unfortunately, this setting led to runtime errors in some of the benchmarks such that we could not use them for the evaluation.

Results. The results of our evaluation are shown in table 2. We performed two runs of measurements, one with the L1 instruction cache enabled and one with disabled L1 instruction cache.

For the measurements performed with activated L1 instruction cache, we give the maximal observed end-to-end execution times of executing the benchmark's `main` function, the result of our analysis when the execution context is ignored, the result of our context-sensitive analysis, the improvement ratio between the later two and the overestimation of the context-sensitive analysis compared to the end-to-end measurements. A smaller ratio denotes a better improvement of the estimated execution time bound when the loop iteration has been taken into account as typical cache behaviour is exploited.

For the measurements performed with disabled L1 instruction cache, we give the maximal observed end-to-end execution time and the result of the context-insensitive analysis. Moreover, we compared them with with the results when the L1 instruction cache is enabled to see what impact the L1 cache has one the execution time of the benchmarks.

On average, an improvement ratio of 0.94 has been reached, i.e. the estimated execution time bound was decreased by 6% when the execution context has been taken into accout. Some benchmarks, like `md5` and `prime` showed much better bound reductions with 33% and 49%, respectively. Other showed almost no improvement at all. On closer inspection, it turned out that those benchmarks had little variance in the observed waypoint execution times. We suspect that the prefetching mechanism of the Cortex-A9 pipeline is able to prevent long delays during instruction fetch.

Table 2 Results for the TACLeBench benchmark suite. The programs have been measured twice, once with L1 instruction cache enabled and once with L1 instruction cache disabled. The first column gives the measured end-to-end execution time in cycles. The second and third columns give the computed execution time estimates, once ignoring the execution context and once taking the loop iteration context into account. The forth column shows the ratio between context-sensitive and context-insensitive estimates (smaller is better). The fifth column shows the overestimation comparing the end-to-end observations and the analysed context-sensitive bounds. The final four columns compare activated and deactivated L1 instruction cache.

	L1 instruction cache activated					L1 instrcution cache deactivated			
program	end-to-end	context-insensitive	context-sensitive	improvement	overestimation	end-to-end	context-insensitive	end-to-end ratio	bound ratio
adpcm_dec	2099113	4292573	3585664	0,84	1,71	8269445	30367476	3,94	7,07
adpcm_enc	52417	90387	88641	0,98	1,69	82542	154810	1,57	1,71
basicmath	19497183	44583249	44138972	0,99	2,26	40107369	168812436	2,06	3,79
binarysearch	1726	2499	2292	0,92	1,33	3237	5739	1,88	2,30
bitcount	149719	466712	463553	0,99	3,10	273131	1599420	1,82	3,43
bitonic	162046	33378706	33361129	1,00	205,87	333355	60556151	2,06	1,81
bsort	2912025	9847181	9829328	1,00	3,38	4470758	15486611	1,54	1,57
complex_updates	7948	11066	10997	0,99	1,38	10735	14086	1,35	1,27
countnegative	98709	256863	255445	0,99	2,59	150774	420206	1,53	1,64
crc	23728	42511	41035	0,97	1,73	1010312	4941180	42,58	116,23
fac	4567	19492	19141	0,98	4,19	10892	54133	2,38	2,78
fft	4967772	2060545070	2060447457	1,00	414,76	7638856	2386497145	1,54	1,16
filterbank	52757627	56175367	56000579	1,00	1,06	130534265	214822229	2,47	3,82
fir2dim	45900	87390	81835	0,94	1,78	91914	174316	2,00	1,99
iir	1779	2401	2227	0,93	1,25	2431	3286	1,37	1,37
insertsort	29397	68977	68291	0,99	2,32	40268	83702	1,37	1,21
jfdctint	37358	39795	39657	1,00	1,06	44769	46902	1,20	1,18
lift	10329419	17278328	13925216	0,81	1,35	20087468	38399195	1,94	2,22
lms	5209670	12704581	12478923	0,98	2,40	8672208	25151604	1,66	1,98
ludcmp	38448	154678	146569	0,95	3,81	65963	256881	1,72	1,66
matrix1	136086	351749	350891	1,00	2,58	275566	537144	2,02	1,53
md5[a]	154573496	615340440	410888084	0,67	2,66	–	1596703270	–	2,59
minver	25746	41736	40152	0,96	1,56	61579	149386	2,39	3,58
pm	178339194	349576605	348784220	1,00	1,96	273680209	503009838	1,53	1,44
powerwindow[b]	8204	–	–	–	–	14096	30889	1,72	–
prime	135944	485956	249622	0,51	1,84	682279	2939746	5,02	6,05
quicksort	56159097	138962109974	135157270647	0,97	2406,69	82844689	206979278894	1,48	1,49
recursion	25991	35506	35506	1,00	1,37	85317	233528	3,28	6,58
st	1726823	2682337	2663482	0,99	1,54	2893966	5864534	1,68	2,19

[a] trace buffer overflow
[b] consistency check failed

Most benchmarks showed reasonable overestimation when comparing the end-to-end execution times and the estimated context-sensitive bounds, with a median of 1.90. Exceptions are `bitonic`, `fft` and `quicksort` which contain data dependent loops and recursions. Since we used the maximal observed loop bounds as bounds for our ILP, we get huge overestimations.

For two benchmarks, we where not able to perform a full evaluation. One is `md5`, where we encountered a trace buffer overflow. We could thus not measure any end-to-end time, but our approach worked nonetheless, as we observed enough small snippets to estimate an overall bound. For benchmark `powerwindow`, we could not give any bound with activated L1 instruction cache because a consistency check in our prototype failed.

Deactivation of the L1 instruction cache leads to a slowdown factor of 3.29 on average. Ignoring the outlier `crc`, which benefits extremely from the L1 instruction cache, the average slowdown factor is 1.94.

Overall, our evaluation shows that the benchmarks benefit from the L1 instruction cache (as visible in the end-to-end measurements), but it is sometimes hard to capture the typical cache behaviour in a hybrid approach which aims for upper bounds.

7 Conclusion and Future Work

In this contribution, we have shown a method that is capable of estimating meaningful WCET of embedded software under the realistic conditions of modern SoCs. Even using the

waypoints instead of basic blocks, a context sensitive aggregation of instruction execution times can be achieved. These execution times can be combined to form a WCET for the overall program (or larger portions of it). Using TACLeBench examples, we can show that the results are highly realistic.

Still many open questions remain. We are currently working on a method to gather a much more detailed statistics of the execution times between waypoints. This would allow a better judgement of the gathered statistics. Also, we want to check, whether this approach can be used for other trace streams like Nexus 5001™. Ultimately, the question still has be answered whether slightly enhanced trace streams could give better results for the WCET estimation.

References

1 AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. `http://www.absint.com/ait/`.

2 ARM Ltd. CoreSight™ Program Flow Trace™ PFTv1.0 and PFTv1.1 Architecture Specification, 2011. ARM IHI 0035B.

3 ARM Ltd. Embedded Trace Macrocell™ ETMv1.0 to ETMv3.5, 2011. ARM IHI 0014Q.

4 ARM Ltd. CoreSight™ Architecture Specification v2.0, 2013. ARM IHI 0029B.

5 ARM Ltd. Embedded Trace Macrocell™ Architecture Specification ETMv4.0 to ETMv4.2, 2016. ARM IHI 0064D.

6 A. Betts and G. Bernat. Tree-based wcet analysis on instrumentation point graphs. In *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*. IEEE Computer Society, April 2006. `doi:10.1109/ISORC.2006.75`.

7 A. Betts, N. Merriam, and G. Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–63. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. `doi:10.4230/OASIcs.WCET.2010.54`.

8 B. Dreyer, C. Hochberger, S. Wegener, and A. Weiss. Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 45–54, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2015.45`.

9 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

10 Freescale Semiconductor, Inc. P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual, Rev. F, 2012.

11 IEEE-ISTO. IEEE-ISTO 5001™-2012, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface, 2012.

12 K. Schmidt, D. Marx, J. Harnisch, and A. Mayer. Non-Intrusive Tracing at First Instruction. SAE Technical Paper 2015-01-0176. `doi:10.4271/2015-01-0176`.

13 S. Stattelmann and F. Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series*

in Informatics (OASIcs), pages 64–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. doi:10.4230/OASIcs.WCET.2010.64.

14 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times – Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

Eager Stack Cache Memory Transfers*

Amine Naji[1] and Florian Brandner[2]

1 **U2IS, ENSTA ParisTech, Université Paris-Saclay, Paris, France**
`amine.naji@ensta-paristech.fr`
2 **LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay, Paris, France**
`florian.brandner@telecom-paristech.fr`

Abstract

The growing complexity of modern computer architectures increasingly complicates the prediction of the run-time behavior of software. For real-time systems, where a safe estimation of the program's worst-case execution time is needed, time-predictable computer architectures promise to resolve this problem. The stack cache, for instance, allows the compiler to efficiently cache a program's stack, while static analysis of its behavior remains easy.

This work introduces an optimization of the stack cache that allows to *anticipate* memory transfers that might be initiated by future stack cache control instructions. These eager memory transfers thus allow to reduce the average-case latency of those control instructions, very similar to "prefetching" techniques known from conventional caches. However, the mechanism proposed here is guaranteed to have no impact on the worst- case execution time estimates computed by static analysis. Measurements on a dual-core platform using the Patmos processor and time-division-multiplexing-based memory arbitration, show that our technique can eliminate up to 62% (7%) of the memory transfers from (respectively to) the stack cache on average over all programs of the MiBench benchmark suite.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems

Keywords and phrases Predictability, Eager Memory Transfers, Stack Cache, Real-Time Systems, Prefetching, Eager Eviction

Digital Object Identifier 10.4230/OASIcs.WCET.2016.5

1 Introduction

The design of modern computer architectures has become more and more complex over the last decades in order to optimize performance and efficiency. In the vast majority of the cases modern architectures try to improve the average-case performance[1] by introducing instruction and data caches, branch predictors, instruction pipelines, and out-of-order execution. The optimizations usually follow the popular design principle: "Make the common case fast." [9]. A downside of this approach is that rare corner cases are often slowed down, which leads to a considerable gap between the best-case and worst-case performance that can be observed. This, in turn, complicates the precise analysis of the timing behavior of real-time programs running on such computer architecture and often results in considerable overestimation.

Time-predictable computer architectures thus gained considerable traction in recent years [15, 10, 12, 11]. In these designs the focus is on predictable and analyzable behavior,

* This work was supported by a grant (2014-0741D) from Digiteo France: "Profiling Metrics and Techniques for the Optimization of Real-Time Programs" (PM-TOP).
[1] Energy consumption recently gained considerable importance as a secondary design goal.

while retaining acceptable average-case performance. The stack cache [2] is an example of a predictable cache design that was shown to be analyzable [6, 1], while efficiently handling memory accesses to stack data [3] at low (hardware) cost. Stack data is cached using a sliding *window* that follows the top of the stack across function calls. The cache is implemented using a ring buffer, following a FIFO policy. Data accesses are, by definition, guaranteed cache hits, the content of the cache thus has to be managed explicitly using three stack cache control instructions: (1) `sres` k allows to reserve k words on the stack, (2) `sfree` k can be used to free previously reserved stack space, and (3) `sens` k, finally, can be used to make sure that at least k words are available in the cache. Only the reserve (`sres`) and ensure (`sens`) operations may initiate time-consuming memory transfers and thus need to be considered during timing analysis [6, 1]. In the case of the `sres` instruction, content might be evicted, or *spilled*, from the cache in order to make space for the k newly reserved words. The `sens` instruction on the other hand might require to *fill* data from main memory when less than k words are available in the cache. The remaining stack cache operations (notably `sfree`) have constant timing and are thus trivial to analyze.

In order to improve predictability and ensure composability, the original stack cache design [2] stalls the processor while performing spilling or filling, even when the stack cache would not be used by any of the subsequent instructions. This allows to analyze the stack cache's timing behavior in isolation from other components of the Patmos computer architecture [12] at the expense of average-case performance. In this work, we explore the use of eager – or anticipatory – memory transfers in order to alleviate this shortcoming. The goal is to improve average-case performance by performing memory transfers in the background alongside with other instructions that are executed by the processor. The eager transfers are, however, not allowed to interfere with the worst-case behavior of the stack cache (or any other hardware component in the system). Most notably, the timing bounds computed for a regular stack cache without our optimizations, should not be invalidated in the presence of our optimizations. This is ensured by exploiting features of a recently proposed stack cache extension [3] to track data that are coherent between the cache and main memory.

2 Background

The stack cache is implemented as a ring buffer with two hardware registers holding pointers [2]: *stack top* (`ST`) and *memory top* (`MT`). The top of the stack is represented by `ST`, which points to the address of all stack data either stored in the cache or in main memory. `MT` points to the top element that is stored only in main memory. The stack grows towards lower addresses. The difference $\texttt{MT} - \texttt{ST}$ represents the amount of occupied space in the stack cache. This notion of *occupancy* is crucial for the effective analysis of the stack cache behavior [6]. In this work we will use an extension of the original stack cache design that allows to track coherent data [3]. This extension introduces a third pointer `LP`, which divides the stack cache into two parts: (1) cache data between `ST` and `LP` is potentially incoherent with the corresponding addresses in main memory, while (2) data between `LP` and `MT` is known to have the same value in the cache and in main memory – the data is coherent. The knowledge about coherent data allows to optimize the stack cache's operation. This is captured by a complementary notion of occupancy, called *effective occupancy* that is given by $\texttt{LP} - \texttt{ST}$.

Clearly, the (effective) occupancy cannot exceed the total size of the cache's memory $|SC|$. The stack cache thus has to respect the following invariants:

$$\texttt{ST} \leq \texttt{MT} \quad (1) \qquad 0 \leq \texttt{MT} - \texttt{ST} \leq |SC| \quad (2) \qquad \texttt{ST} \leq \texttt{LP} \leq \texttt{MT} \quad (3)$$

Stack Cache Operations: The stack control instructions manipulate the three stack pointers and initiate memory transfers to/from the cache from/to main memory, while preserving the above equations. We summarize these instructions below, details are in [2, 3]:

`sres` k: Subtract k from `ST`. If this violates the Equations from above, data has to be evicted from the cache. In the simplest case only coherent data is discarded, i.e., $\texttt{LP} - \texttt{ST} \leq |SC|$ but $\texttt{MT} - \texttt{ST} > |SC|$. It then suffices to set $\texttt{MT} = \texttt{ST} + |SC|$. Otherwise, a memory *spill* of incoherent data has to be performed by a transfer covering the address range $[\texttt{ST} + |SC|, \texttt{LP}]$ to main memory. When spilling is completed, the `LP` and `MT` pointers are updated $\texttt{LP} = \texttt{MT} = \texttt{ST} + |SC|$.

`sfree` k: Add k to `ST`. If this violates Equation 1 or 3, `MT` and/or `LP` are simply set to `ST`. Main memory is not accessed.

`sens` k: Ensure that the occupancy is larger than k. If this is not the case, a *fill* from main memory is initiated covering the address range $[\texttt{MT}, \texttt{ST} + k]$. `MT` is subsequently incremented to $\texttt{MT} = \texttt{ST} + k$. `LP` does not change.

Data in the cache is accessed using dedicated stack load and store instructions. These instructions only access the stack cache's ring buffer and thus exhibit constant execution times. This is particularly true for stack store instructions, which only modify the cached value but not the backing main memory. Modified values are potentially transferred to main memory only by `sres` instructions. This policy resembles traditional *write back* caches. Also note that the `LP` might be updated by stack store instructions, i.e., when previously coherent data is modified. This has no impact on the constant instruction latency [3].

Compiler Support: The compiler manages the stack frames of functions quite similar to other architectures with exception of the ensure instructions. Stack frames are typically allocated upon entering a function (`sres`) and freed immediately before returning (`sfree`). A function's stack frame might be (partially) evicted from the cache during calls. Ensure instructions (`sens`) are thus placed immediately after each call. Evicted data is then reloaded into the cache. Functions may only access their own stack frames. Data that is larger than the stack cache or that is shared is allocated on a *shadow stack* outside the stack cache.

3 Eager Memory Transfers

Prefetching is a well-known technique used in conventional caches, which aims to hide memory access latencies caused by cache misses. Instead of waiting for a cache miss to initiate a memory transfer, prefetching anticipates such misses and fetches data from memory in advance of the actual memory reference. The idea, though simple, raises two important problems: (1) the addresses of future memory references need to be predicted and (2) side effects may arise due to the eviction of data from the cache in order to make space. Both of these issues are difficult to solve in general settings and pose even more problems in the context of real-time systems requiring predictability.

We explore the use of *eager memory transfers* – combining prefetching and eager eviction [8] – in order to reduce the latency of the stack cache control instructions. We introduce two kinds of eager memory transfers: (1) *eager spilling* transfers data from the stack cache to main memory, while (2) *eager filling* transfers data from main memory to the stack cache. The stack cache, in contrast to conventional caches, tracks its content using simple pointers and thus can only cache a contiguous memory region between the `ST` and `MT` pointers. In the following we will exploit this feature in order to realize "prefetching-like" functionality for the stack cache and address the two aforementioned problems faced in standard caches.

Address Prediction: Due to the use of pointers to track the stack cache content, it is trivial to predict the address of any future memory transfers that might be initiated by any stack cache control instruction. Data is either read from memory at the address starting at `MT` or written to memory at the address up to `LP`, depending on whether the (effective) occupancy will grow too large (`sres` spilling up to `LP`) or will become too small (`sens` filling from `MT`). It thus suffices to predict whether data needs to be spilled or filled with regard to the future stack cache control instructions.

Side effects: We rely on a recently proposed stack cache extension [3] that allows to track coherent data between the stack cache and main memory in order to avoid side effects when performing eager memory transfers. A first observation is that eager spilling only needs to consider incoherent data (just like regular spilling). The eagerly spilled data is, however, not evicted from the stack cache. Instead, it simply becomes coherent. Since no data was evicted from the cache, side effects on future `sens` instructions are excluded. Similarly, since the amount of incoherent data was reduced, the spilling at future `sres` instructions is potentially reduced. A second observation is that eagerly filled data is known to be coherent. Side effects on future `sres` instructions are consequently excluded after eager filling since the amount of incoherent data did not change. The filling at future `sens` instructions, on the other hand, is reduced due to the newly loaded data.

The eager memory transfers are guaranteed to have no side effects on the stack cache itself. However, side effects on other hardware components, and here in particular the bus and main memory, may arise. For instance, a cache for regular data might be blocked by an eager memory transfer upon a cache miss. Such interferences may, of course, impact the program's worst-case performance and compromise predictability as well as composability.

An elegant solution is to exploit the arbitration scheme that mitigates between competing memory accesses [4, 1]. In the context of this work, we use the Patmos multi-core architecture, which relies on time-division multiplexing (TDM) to arbitrate main memory accesses. In the following we assume that each processor core may transfer a single memory burst from/to main memory in a dedicated *TDM slot*. Transfers may only be initiated at the beginning of a TDM slot, which are periodically scheduled in a *TDM period*. The duration of a period then depends on the number of cores n and the duration of a TDM slot k and is given by $n \cdot k$ cycles. We assume that the memory controller is able to process transfers with arbitrary start addresses and lengths. The actual memory transfer is, however, performed at the granularity of bursts, i.e., the start address and length are aligned accordingly to the burst size (excess data is either masked or discarded). In such a setting it is easy to detect TDM slots that are not used by any other hardware component. It suffices to check that no other memory request is pending at the beginning of the processor core's TDM slot. The free TDM slots of a processor can then be used to perform the eager memory transfers and avoid any side effects on either the stack cache itself nor any other hardware component.

3.1 Eager Fill

The *eager fill* operation aims to reduce the latency of a future ensure instruction `sens` k. Recall that filling is required only when the occupancy is too small, i.e., $\texttt{MT} - \texttt{ST} < k$. The occupancy has to be increased in order to reduce the latency. This can be achieved by loading, i.e. filling, data from main memory such that `MT` can be pushed upwards until the occupancy reaches the stack cache size. The corresponding memory transfer, however, has to be limited to a single burst transfer in order to guarantee that only a single TDM slot is occupied. Assuming a burst size `BS`, an eager fill operation thus proceeds as depicted by the

```
if (MT - ST < |SC|) {
  start = MT;
  end = ⌊(MT+BS)/BS⌋ × BS;
  fill(start, end);
  MT = end;
}
```

(a) The eager fill operation.

```
if (LP − ST < k) {
  end = LP;
  start = ⌈(LP−BS)/BS⌉ × BS;
  spill(start, end);
  LP = start;
}
```

(b) The eager spill operation.

Figure 1 Pseudo code illustrating the operation of the eager filling and eager spilling.

algorithm in Figure 1a. The eager fill operation can be initiated whenever a TDM slot is free and is then guaranteed to be free of any interference with other hardware components that might wish to access main memory. It remains to show that the worst-case timing of subsequent stack cache operations is not affected. Three cases have to be considered, depending on the kind of the next stack cache control instruction:

`sres` k: May only initiate a memory transfer when incoherent data has to be evicted from the cache. The address range of the transfer ($[\texttt{ST}+|SC|, \texttt{LP}]$) only depends on the position of `ST` and `LP`. Eager filling does not modify either of those pointers (effective occupancy) and thus cannot impact spill costs.

`sfree` k: Free instructions do not access memory and exhibit constant latency.

`sens` k: May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST}+k]$) only depends on `ST` and `MT`. The former is not impacted by eager filling, while the address of `MT` is incremented, i.e., the occupancy was previously increased. Fill costs thus may only be reduced.

Eager fill operations, consequently, may only improve the latency of future `sens` instructions. Note, however, that some side effects may still arise. This may appear when all filling of an `sens` instruction is eliminated. In this case, the `sens` instruction no longer synchronizes with the TDM period and may change the alignment of subsequent memory accesses. This may incidentally increase the number of stall cycles of these memory accesses. The number of additional stall cycles can, however, never exceed the gain induced by eager filling. WCET estimates computed without considering eager filling thus remain valid.

3.2 Eager Spill

The aim of the eager spill operation is to anticipate and reduce future spill costs associated with subsequent `sres` instructions. A spill is initiated by an `sres` if the effective occupancy would exceed the size of the stack cache, i.e., $\texttt{LP} - \texttt{ST} > |SC|$. The effective occupancy thus has to be lowered in order to reduce the spill latency. One possible solution is to copy incoherent stack data to main memory without evicting them from the cache. This allows to decrement `LP` and thus reduce the effective occupancy.

As for eager filling, the corresponding memory transfer size must not exceed the burst size so that at most one TDM slot is used. Assuming a burst size `BS`, an eager spill operation then proceeds as depicted by Figure 1b. The eager spill operation can be performed during free TDM slots as soon as the effective occupancy is non null. We will, nonetheless, prevent the spilling of data from the stack frame of the current function. This is because it may happen that data about to be eagerly spilled is modified by a stack store instruction. This would require additional checks to ensure that incoherent data is correctly tracked and increase hardware costs as well as complexity. As before, only free TDM slots are used,

which guarantees that eager spill operations cannot interfere with other memory accesses. The worst-case timing of subsequent stack cache control operations is also not affected:

`sres` k: May only initiate a memory transfer when the effective occupancy becomes too large. The covered address range ($[\texttt{ST} + |SC|, \texttt{LP}]$) only involves the `ST` and `LP` pointers. The latter is lowered by eager spilling, while the former is not modified, i.e., effective occupancy was previously decreased. The spill costs experienced by an `sres` instruction thus may only be reduced.

`sfree` k: Free instructions do not access memory and exhibit constant latency.

`sens` k: May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST} + k]$) only depends on `ST` and `MT`. Both are not impacted by eager spilling. Fill costs thus cannot be impacted by eager spilling.

Eager spill operations, consequently, may only improve the latency of future `sres` instructions. Similarly to eager filling, the alignment of memory accesses with regard to the TDM period may change. The worst-case timing behavior of the program is not impacted.

3.3 Spill/Fill Arbitration

The eager fill and spill operations can be executed asynchronously alongside other instructions that are executed by the processor whenever a free TDM slot is encountered and the respective conditions necessary to perform a transfer are met. The two operations naturally compete for the available TDM slots, we thus defined several simple arbitration policies.

Spill/Fill-Only: As the names indicate, in these two configuration schemes only one of the two eager operations is performed throughout program execution, subject to the respective conditions as described above. This allows to quantify the attainable profit of either operation, ignoring the potential overhead induced by unprofitable eager transfers.

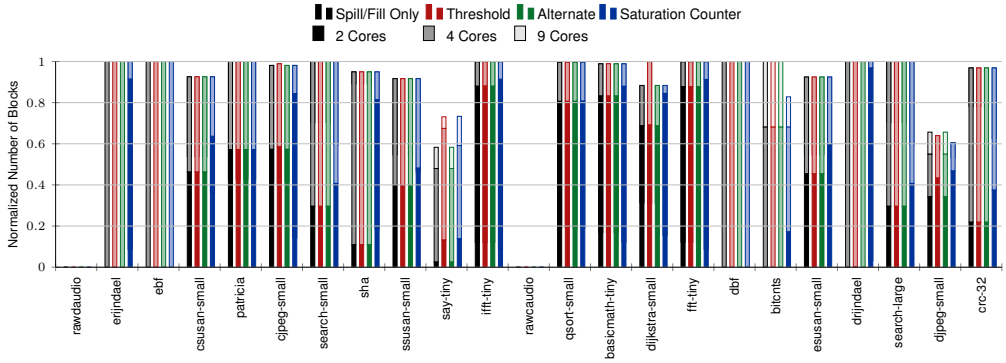
Alternate: Eager spill and fill are performed alternatingly in order to attain the maximum profit by applying both operations whenever this is possible on a fair arbitration policy.

Threshold: This approaches aims to reduce the amount of unprofitable eager operations, e.g., eagerly spilling data that is never evicted. Eager operations are performed alternatingly until a preset (effective) occupancy level (threshold) is reached. In the experiments, eager spilling stops when the effective occupancy is half of the stack cache size. Likewise, eager filling stops when the occupancy reaches half of the stack cache size.

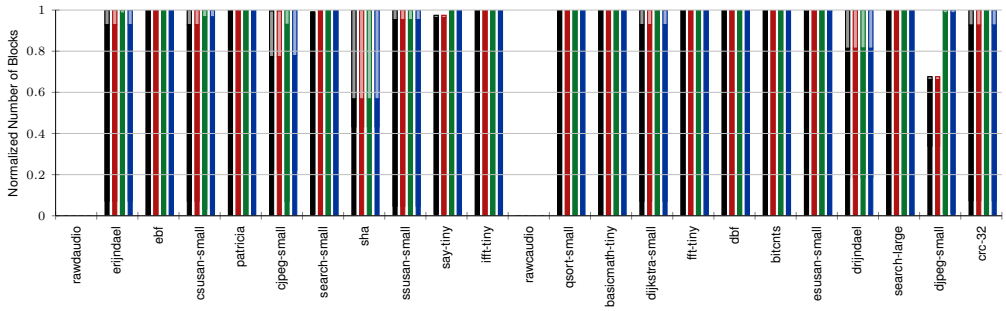
Saturation Counter: In this approach, the kind of the next stack control instruction is predicted and eager operations chosen such that its transfer costs are reduced. The hypothesis is that `sres` and `sens` instructions are performed in sequences when descending/ascending the call chain. The prediction uses a saturation counter, similar to branch prediction [9], that is in-/decremented up to prespecified maximum levels whenever an `sres`/`sens` instruction is encountered. The eager spill/fill operations are then only permitted when the counter value lies within predefined ranges. We use a simple 1-bit saturation counter in the experiments.

4 Experiments

We evaluated eager memory transfers using the cycle-accurate simulator of the Patmos processor [12], which implements a stack cache and its associated control instructions. It



(a) Spill. The black bar represents the Spill-Only configuration.

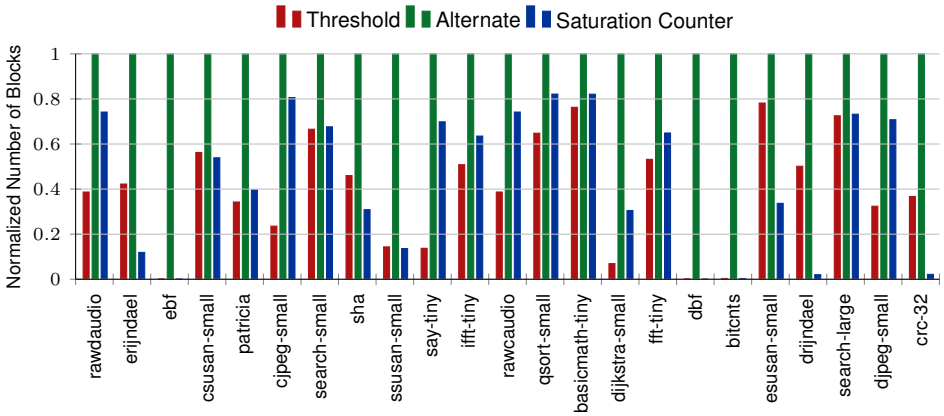


(b) Fill. The black bar represents the Fill-Only configuration.

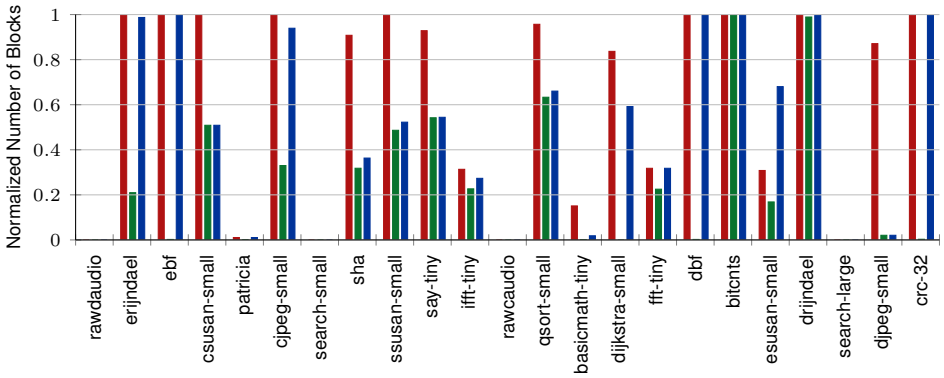
Figure 2 Normalized number of total cache blocks regularly spilled/filled with respect to standard stack cache implementation supporting lazy pointer. (Lower is better).

also allows to simulate several processor cores in parallel that access a shared main memory using bursts of 32 B. Memory arbitration is then performed using a TDM policy. We furthermore extended the stack cache implementation to support eager memory transfers using the arbitration strategies described above. Benchmarks of the MiBench benchmark suite [5] were compiled using optimizations (`-O2`) and subsequently executed on multi-core configurations with 2, 4 (2×2), and 9 (3×3) cores. Each core is equipped with a 256 byte stack cache, a 64 KB, 4-way set-associative data cache using a *least-recently used* replacement and write-through policy, as well as a 64 KB, 64-entry method cache using *first in, first out* replacement. The stack cache operates on 4 byte blocks, while the block size of the other caches matches the burst size of the main memory. Memory accesses take 21 cycles.

Figure 2 shows the normalized reduction in the number of blocks spilled and filled by `sres` and `sens` instructions in comparison to regular program execution without eager memory transfers. For eager spilling, results show a considerable reduction of spill costs by 62% over all benchmarks for the dual-core platform. For several benchmarks all spilling is performed by the eager operation (`erijndael`, `ebf`, `dbf`, `bitcnts`, `drijndael`). The total stack size of `rawcaudio` and `rawdaudio` fits into the stack cache. So, no spilling is ever performed for these benchmarks. The results for 4 and 9 cores are very close and give reductions of 6% and 1% respectively. Notable differences can be observed for `say-tiny`, `bitcnts`, and `djpeg-small`. This can be explained by the increased TDM period, which reduces the number of free TDM slots and the potential to perform eager memory transfers. All arbitration strategies were able to reduce the number of blocks spilled by `sres` instructions. The Alternate and Threshold configurations performed best and almost always reached the best possible result represented by the Spill-Only strategy.



(a) Spill.



(b) Fill.

Figure 3 Efficiency of the various eager spill/fill arbitration policies relative to the Spill- and Fill-Only configurations on a dual-core platform (Lower is better).

The results for eager filling are less pronounced, resulting in reductions of only 7.4%, 1.7%, and 0.1% for the platforms with 2, 4, and 9 cores respectively. The large difference with eager spilling is surprising. Investigations showed that our hypothesis that `sres`/`sens` instructions often appear in sequences appears to hold. However, the average distance between `sres` instructions is typically much larger than the distance between `sens` instructions. The probability to encounter free TDM slots thus is much smaller between consecutive `sens` instructions, thus reducing the amount of eager filling that can be performed. Again, all strategies are able to achieve reductions. However, the Threshold configuration clearly performs best. This is once more surprising, since the theoretical bandwidth available for filling in the Alternate approach should at least reach 50% of the bandwidth of the Threshold configuration. It appears that the limited number of TDM slots available in between `sens` instructions aggravates the competition with eager spilling, explaining this bias. Further investigations are, however, needed to confirm this hypothesis.

We also performed measurements on a single-core configuration, where the processor performs memory accesses using a private bus (without TDM). Eager operations were initiated following the Alternate arbitration scheme immediately when no other bus requests were pending. Note that in this case interferences with other memory accesses frequently occur. We observed that spilling *and* filling of the stack control instructions was completely eliminated for almost all benchmarks, i.e., all cache transfers were carried out by eager operations. This indicates that eager transfers are effectively limited by the number of free

TDM slots. An interesting idea would thus be to investigate means to explicitly allocate non-free TDM slots to eager operations. This could allow to entirely eliminate stalls at the stack control instructions in an analyzable and predictable manner.

In addition to the effective reduction by the various configurations in the number of memory transfers suffered by `sres` and `sens` instructions, we also compared the relative efficiency of the approaches. Figure 3 shows the normalized number of blocks eagerly spilled/filled with respect to the aggressive Spill-Only and Fill-Only configurations respectively. The Threshold configuration appears to provide the best trade-off between efficiency and the actual reduction of memory transfers by the stack control instructions. On the dual-core platform and over all benchmarks, it eagerly spills 60% and eagerly fills 30% fewer cache blocks than the Spill-/Fill-Only configurations respectively. Still the amount of excess spilling (and to a lesser degree filling) is considerable. On average, over all benchmarks 75 times the number of cache blocks are spilled compared to the number of cache blocks spilled by the program when eager memory transfers are deactivated.

However, excess spilling is not necessarily a waste. The reduced effective occupancy may reduce the cost of context switching [1]. The Threshold configuration on a dual-core platform decreases the average effective occupancy over the benchmarks' entire execution time by about 25%. For `ssusan_small`, for instance, the reduction amounts to 68%, thus considerably reducing the context switch cost related to the stack cache.

5 Related Work

Prefetching data before it is needed is a common concept in computer science and particular in computer architecture design [13]. However, the vast majority of prefetching mechanisms are *only* designed to improve the average-case and thus are not suited for the use in real-time systems. The notable exception is the WCET-preserving stream prefetcher proposed by Garside and Audsley [4]. The approach avoids side effects on the content of the cache by introducing separate prefetch buffers – similar to the initial work on stream prefetching by Jouppi [7]. In addition, properties of the bus arbitration scheme are exploited to schedule "prefetch slots". The authors observe that the interference between multiple cores in the system is typically overestimated. A prefetch can thus be scheduled whenever an interference is *overdue*, while respecting the worst-case execution time of the program. The actual implementation is based on a fixed-priority scheme with a predefined blocking factor to avoid starvation. The approach provides excellent average-case improvements. However, it appears difficult to improve the WCET estimation by considering the prefetching, due to the potential interaction with all other cores in the system. Our approach does not require a separate memory structure and directly operates on the stack cache. An address prediction mechanism is also not required since addresses are a priori kown (`MT` or `LP`). We thus expect a much simpler hardware design. Instead of a fixed-priority scheme we rely on free TDM slots that are *left over* by the program. Interference from other programs or processor cores with regard to the eager memory transfers are consequently excluded. It thus appears feasible to actually improve WCET estimates by taking the eager memory transfers into consideration.

In addition to prefetching, data is also transferred from the stack cache to main memory by eager spill operations. To the best of our knowledge such a mechanism was not yet proposed in the context of time-predictable cache design. Similar ideas were, however, explored for conventional write-back cache designs and termed *eager write-back* [8].

An alternative approach is to allocate code as well as data to scratchpad memories [14]. However, scratchpad memories typically complement caches instead of replacing them. The

stack cache mixes properties of both, conventional caches and scratchpads, and thus is situated in between those concepts. Due to space considerations we do not elaborate these techniques in more depth here.

6 Conclusion

We presented an elegant and simple extension of the stack cache that allows to perform memory transfers eagerly in order to reduce the latency of future stack cache control instructions. We exploit the capability to track coherent data in the stack cache using the lazy pointer (`LP`), which allows us to distinguish between the effective occupancy and the total cache occupancy. Eager filling increases the occupancy and thus may profit future `sens` instructions, while eager spilling decreases the effective occupancy and thus may profit `sres` instructions. The interplay between effective occupancy and occupancy guarantees that the worst-case timing is not impacted. In addition, we propose to perform these eager operations in free TDM slots to avoid any interference with concurrent memory accesses.

References

1 S. Abbaspour, F. Brandner, A. Naji, and M. Jan. Efficient context switching for the stack cache: Implementation and analysis. In *Proc. of the Int'l Conf. on Real Time and Networks Systems*, RTNS'15, pages 119–128. ACM, 2015. doi:10.1145/2834848.2834861.

2 S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013.

3 S. Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92, 2014.

4 J. Garside and N. C. Audsley. WCET preserving hardware prefetch for many-core real-time systems. In *Proc. of the Int'l Conf. on Real-Time Networks and Systems*, RTNS'14. ACM, 2014. doi:10.1145/2659787.2659824.

5 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workshop on Workload Characterization*, WWC'01, 2001.

6 A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proc. of the Conf. on Real-Time Networks and Systems*, RTNS'13, pages 55–64, 2013. doi:10.1145/2516821.2516828.

7 N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the Int'l Symp. on Computer Architecture*, ISCA'90, pages 364–373. ACM, 1990. doi:10.1145/325164.325162.

8 H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback – a technique for improving bandwidth utilization. In *Proc. of the Int'l Symp. on Microarchitecture*, MICRO 33, pages 11–21. ACM, 2000. doi:10.1145/360128.360132.

9 D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4rd edition, 2012.

10 J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conf. on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.

11 C. Rochange, S. Uhrig, and P. Sainrat. *Time-Predictable Architectures*. Wiley, 2014. doi:10.1002/9781118790229.

12 M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: the Patmos approach. In

Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France, volume 18, pages 11–21. OASICS, 2011. doi:10.4230/OASIcs.PPES.2011.11.

13 A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982. doi:10.1145/356887.356892.

14 V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proc. of the Int'l Real-Time Systems Symp.*, RTSS'05, pages 223–232. IEEE, 2005. doi:10.1109/RTSS.2005.45.

15 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.

The Variability of Application Execution Times on a Multi-Core Platform*

Vincent Nélis[1], Patrick Meumeu Yomsi[2], and Luís Miguel Pinho[3]

1 **CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal**
nelis@isep.ipp.pt
2 **CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal**
pamyo@isep.ipp.pt
3 **CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal**
lmp@isep.ipp.pt

Abstract

It is a known fact that processes running concurrently on different cores in a multicore environment interfere with each other on the processor shared resources. The contention on these shared resources considerably slows down the execution on every core since sometimes the cores must stall while their requests to access the resources are being served. But by how much the execution may be slowed down due to this interference? In this paper we answer this question with numbers coming from experimentation. That is, we quantify the magnitude of the impact of the interference on the execution time by running programs taken from the TACLeBench benchmark suite, a popular benchmark suite in the real-time research community, on the first generation of Kalray manycore processor family, the MPPA-256 (the development board) that goes by the codename "Andey".

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

Keywords and phrases Execution time variability, timing analysis, WCET estimates, multi-cores, many-cores

Digital Object Identifier 10.4230/OASIcs.WCET.2016.6

1 The problem of inter-core interference

Determining the worst-case execution time (WCET) of a software application has always been a major problem in the design of real-time systems. Those WCET estimates are at the base of the whole stack of higher-level analyses defined to characterize the timing behaviour of the system and verify its timing requirements. Computing estimates that are as close as possible to the actual maximum execution time is crucial. Under-estimating the application execution times during the analysis phase may result in designing an over-utilized system that does not meet its timing requirements, whereas over-estimating them may result in an over-dimensioned (costlier) system of which the resources are under-utilized and thus wasted.

In multi-core architectures, the problem of finding WCET upper-bounds is further exacerbated by the high number of resources shared between the cores. In such platforms running processes may execute concurrently on different processor cores but any of their

* This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement no. 611016 (P-SOCRATES).

accesses to a memory (to fetch an instruction or data) traverses multiple layers of arbitration in which the request may contend with others, emitted by processes running on other cores. Contrasting with single-core architectures, on multi-cores the time during which a core stalls waiting for a memory request to be served is a significant component of the overall execution time of a program.

The research community has addressed the additional problem of estimating the time-penalty caused by inter-core interference in various ways. One methodology consists in estimating the worst-case interference that a program may incur at runtime and inflate the individual WCET upper-bounds accordingly. This preserves the original analysis flow used in single-core systems, in which individual WCET are estimated for every software program and then fed as input into the higher-level schedulability analyses. Other works take into consideration that the interference between processes effectively depends on the scheduling decisions taken for these processes. In those works the estimated maximum interference is accounted for at the schedulability level. Other initiatives acknowledge that the interference between processes may be way too high on multi-cores and thus focus on how to temporally isolate the processes to nullify, or at least mitigate, the interference between them.

Related work and contribution: It is important to understand that the present work does not aim at measuring or determining the *worst-case* execution time of a given program. For that particular problem, the interested reader may consult [6] for an overview of the state-of-the-art solutions. Broadly speaking there are three main methodologies to estimate the worst-case execution time of a software program: static, measurement-based, and hybrid analyses, with their respective advantages and drawbacks discussed in [4]. More recently, the research community has been focusing on statistic methods as well (see [3] for an interesting starting point). In this work we performed a set of experiments to *measure and quantify the variability in the execution time of a program due to resource contention when run concurrently with other programs on a multi-core processor*. That is, we like to point out the magnitude of the time-overhead due to the interference on the processor shared resources. To do so, we have run a collection of programs taken from various application benchmark suites on the Kalray MPPA-256 development board. We have run the selected programs numerous times, always over the same set of input data, and varied the execution conditions.

2 Our test-case platform and applications

Platform settings: Our experiments have been carried out on the first generation of Kalray manycore processor family, the MPPA-256 that goes by the codename "Andey" [2] (we used the development board). It is a clustered many-core platform composed by an array of 16 clusters and 4 I/O subsystems, themselves connected by two NoCs (illustrated in Fig. 1). Cores are grouped in clusters connected by a Round-Robin (RR) arbitrated Network-on-Chip (NoC) in a 2D-torus topology [1]. Each cluster contains 17 identical VLIW cores: 16 *compute cores* that are dedicated to general-purpose computations and one *Resource Manager* (RM) core whose responsibility is to manage the processor resources on the behalf of the entire cluster (maps and schedules threads on compute cores) and organize the communication between its cluster and the other clusters, as well as with the main off-chip memory. Every compute core has a private instruction and data cache. All the cores are fully timing compositional [7] in the sense that they do not exhibit timing anomalies. Additionally each cluster also contains a Debug Support Unit, a network interface for receiving data requests from the Data-NoC and a DMA engine used for data transmission over the Data-NOC.

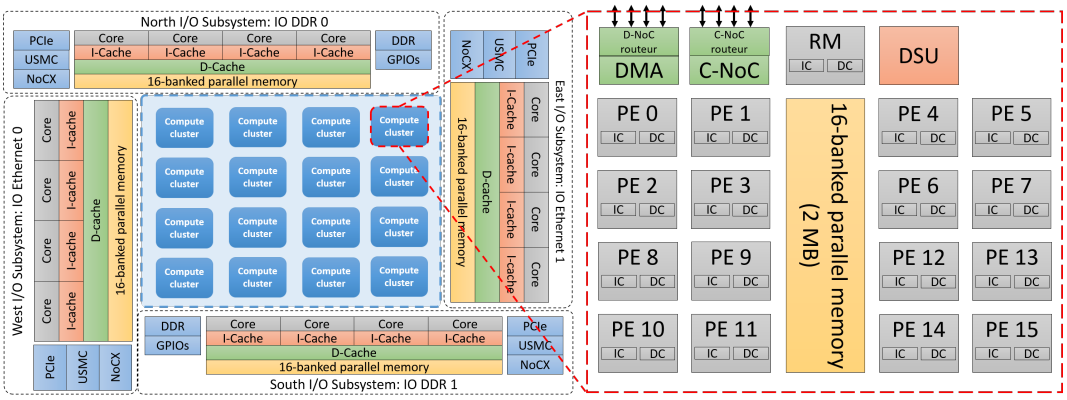


Figure 1 Outline of the architecture of the Kalray MPPA-256 (Andey) manycore processor.

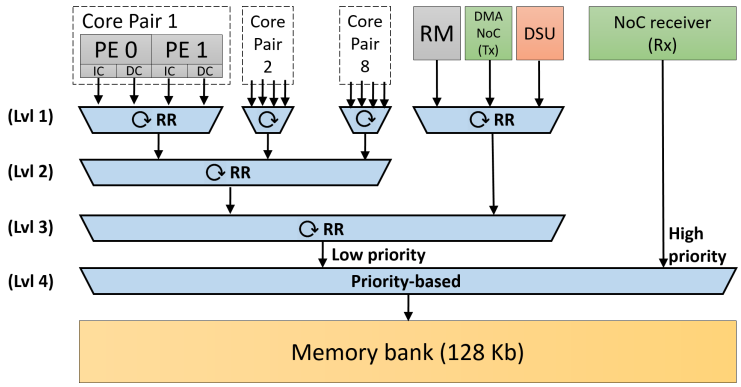


Figure 2 Memory request arbitration of the Kalray MPPA-256.

Regarding the organization of the memory subsystem, each cluster has a local shared memory comprising 16 banks, each with a capacity of 128 KB, for a total memory capacity of 2 MB per cluster. The memory banks are organized into two groups of equal sizes, referred to as the banks on the *left* and *right* sides of the memory, respectively. Similarly, the 16 compute cores are grouped into 8 *core-pairs*[1]. Although the cluster address space can be divided among banks in an interleaved fashion (useful for high-performance and parallel applications), this work uses the blocked memory mode where the address space is divided in a sequential manner. This results in more predictable system behavior, which is a desired characteristic in systems subject to timing requirements.

The arbitration of memory requests is performed in four levels (stages), as depicted in Fig. 2. The first three levels use the Round-Robin arbitration scheme. The first level arbitrates memory requests issued from the two data caches and instruction caches of each core-pair. At the second level, the requests issued from each core-pair compete against the requests coming from the other core-pairs. At the third level, requests from all core-pairs compete against requests from the RM, the DSU, and the DMA. Finally, at the fourth level, the scheduled requests compete with those coming from the D-NOC (Rx) under static-priority

[1] This organization in core-pairs is specific to the "Andey". The second generation of Kalray's manycore processor family, the MPPA2 high-speed I/O processor (codename "Bostan") dropped this architectural choice.

Table 1 The 13 benchmark programs for which we measured the execution time. "LoC" is the total number of lines of code of all source code files belonging to a benchmark, empty lines and comments are not counted. This information as been taken as is from [5].

Name	Description	LoC	Origin
ammunition	C compiler arithmetic stress test	2508	misc
cjpeg_jpeg6b_transupp	JPEG image transcoding routines	1599	MediaBench
cjpeg_jpeg6b_wrbmp	JPEG image bitmap writing code	1296	MediaBench
dijkstra	All pairs shortest path	227	MiBench
epic	Efficient pyramid image coder	994	MediaBench
gsm_decode	GSM 06.10 provisional standard decoder	1368	MediaBench
gsm_encode	GSM 06.10 provisional standard encoder	1940	MediaBench
h264dec_ldecode_block	H.264 block decoding functions	1574	MediaBench
mpeg2	MPEG2 motion estimation	1533	MediaBench
ndes	Complex embedded code	407	MRTC
rijndael_decoder	Rijndael AES decryption	3043	MiBench
rijndael_encoder	Rijndael AES encryption	1024	MiBench
statemate	Statechart simulation of a car window lift control	1053	MRTC

arbitration, where requests from the NoC always have a higher priority. Note: in order to minimize contention, the second, third, and fourth levels of arbitration are replicated for each memory bank. The first level is only duplicated for memory banks located on the left and the right side of the memory, respectively, so that paired cores can access banks on different sides in parallel without interfering with each other. From the organization of the memory and its four-stages arbitration mechanism, it is easy to discern that there can be substantial interferences arising from different sources, e.g., from the concurrent accesses to memory banks from different cores or the contention on the NoC for the access to off-chip memory.

Application test-cases: We measure the execution time of a set of programs taken from the TACLeBench benchmark suite (from those labelled as "sequential benchmarks"). TACLeBench provides a freely available and comprehensive benchmark suite for timing analysis, featuring complex multi-core benchmarks [5]. We selected 13 programs out of the 102 programs available in the TACLeBench suite (see Table 1). Those programs are provided as ANSI-C 99 source codes that are 100% self-contained, i.e. no dependencies to system-specific header files via "#include" directives (eventually used functions from math libraries are also provided in C source code) [5].

3 Our approach to measuring the execution times

Our timing analysis methodology is based on the intuitive idea that the total execution time of any piece of code, e.g. a basic block, a software function, or an entire application, can be seen as composed of two main terms: the "intrinsic" time spent executing every instruction of the code and the time spent waiting for a shared software or hardware resource to become available. It is fundamental to clearly understand the difference between these two components.

The maximum intrinsic execution time (MIET): For a given set of input data, it is the time that the program takes to produce *the* corresponding output[2], *assuming that all software and hardware services provided by the execution environment and shared among different cores are always available* (the core running that program never stalls waiting for one of these resources to become available). That is, the intrinsic execution time of a program is its execution time when it runs in isolation, i.e. with no interference whatsoever with the rest of the system on the shared resources. Note that it does not mean that the code and data of the program are preloaded in the caches before execution, rather it means that wherever the information is stored, there will be no interference when fetching it.

On an "ideal" hardware architecture every instruction should take a constant number of cycles to execute (i.e. there is no time variation what-so-ever) and thus running the same program in isolation over the same set of input arguments always results in the exact same execution time. Although this may sound like a very strong assumption to make in practice, we will see that on a platform such as the Kalray MPPA-256 this assumption is reasonable. By running a preliminary set of tests with the same program an arbitrary number of times over the same inputs, we experienced a variation of its execution time of typically less than 0.1% of the maximum observed.

The maximum extrinsic execution time (MEET): For a given set of input data, it is the time that the program takes to produce its output *assuming a maximum interference on all the shared resources*. That is, the extrinsic execution time of a function is its execution time assuming that all the software and hardware services provided by the execution environment and shared among the cores are constantly saturated by requests from other system components. As we will see, contrary to the intrinsic execution time, the extrinsic execution time is generally subject to substantial variations due to the high number of processor resources shared amongst software functions.

3.1 Extraction of the MIET: the isolation mode

In order to extract the MIET of an application, the platform is configured in what we call the "isolation mode": the entire application is assigned to a single thread that is pinned to a core and all the other cores are shut down and kept idle. This is done to minimize the interference with the rest of the system. To enforce this mode of execution, we have implemented a platform-specific API for the Kalray MPPA-256. This API provides a set of functions and global parameters to perform the following tasks:

- Enforce that the thread running the analyzed program is executed, uninterruptedly, on a single core,
- Synchronise the IO cores and the cluster cores so that it is guaranteed that nothing runs in the background that could interfere with the execution of the analysed program, and
- Perform additional operations on the demand of the user to [re]configure the cluster before processing. Specifically, any of the following actions, or a combination of them can be performed: (a) Activate/deactivate the data cache; (b) Invalidate the data cache; (c) Flush

[2] We assume that there is no functional random behaviours involved in the definition of the analysed program. That is, the outcome of evaluating a condition is never the result of an operation involving randomly-generated numbers. Under this assumption of not involving randomness in the control flow of the program, running it multiple times over a same set of input data always results in taking the same path throughout the program's code and thus execute the exact same sequence of instructions and eventually, it always produces the same output.

the data cache; (d) Change the operating mode of the data cache (i.e. make the cores stall on access or not); (e) Invalidate the Data-TLB; (f) Activate the instruction cache; (g) Invalidate the instruction cache; (h) Set the address mapping scheme of the 2MB shared memory in each cluster (i.e., set the address mapping scheme to "interleaved" or "sequential" mode).

With these configuration options, we define two different configurations of the platform in order to give a hint of the type of results that can be expected from applications with very different memory access profiles:

- **The High-Performance (HP) configuration.** In this configuration, the data cache is enabled; The content of the data cache is neither invalidated nor flushed before each execution; The "stall-on-access" mode is disabled (that is, the core does not stall while waiting for a data to be fetched); The content of the data TLB is not invalidated before each execution; The instruction cache is enabled; The content of the instruction cache is not invalidated before each execution; and the 2MB shared memory of the cluster is set in "interleaved" mode, to allow data to span several banks.
- **The Low-performance (LP) Configuration.** In this configuration, the data cache is disabled; The "stall-on-access" mode is enabled; The instruction cache is disabled; and the 2MB shared memory of the cluster is set in "sequential" mode, to allow data to span multiple adjacent banks if and only if it does not fit in the bank currently in use.

Clearly, using the LP or HP configuration has a substantial impact on the execution time of the application. The reason for defining these two platform configurations is not to assess the level of performance that is achievable in general on the Kalray-MPPA 256. Rather, we want to give a hint at the type of results that can be expected from applications with very different memory access profiles. To understand this relation between the platform settings and the memory access pattern of the application, it is important to understand that if the memory footprint of the analyzed program (instruction + data) is small enough, it will fit entirely in the private cache of the core on which the application is run. Therefore, when executing the program using the HP configuration, the instructions and data will be loaded once in the private cache and the program will not need to communicate further with the shared memory. That is, running a program with a small memory footprint using the HP configuration is equivalent to running a program with very limited communication with the shared memory. As it will be seen, when using the HP configuration the execution time of some of the benchmark programs used in our experiments is not, or almost not, affected by the execution of other programs running concurrently on other cores. On the contrary, when the LP configuration is used, since the caches are disabled, the analyzed program must frequently communicate with the shared memory during its execution. As a result, it is way more subject to interference with other programs running concurrently on the other cores and accessing the shared memory as well.

3.2 Extraction of the MEET: the contention mode

In order to extract the MEET of each application, the platform is configured in what we call the "contention mode". In this mode, we start each application and try to interfere as much as possible with its execution while it is running. The objective of the contention mode is to create the "worst" execution conditions for the application so that its execution is constantly suspended due to interference with other programs. This gives us an estimation of the maximum execution time of the application when it suffers maximum interference from other programs on the shared resources.

The contention mode is similar to the isolation mode in that the analyzed program is assigned to a single thread that is pinned to a core (here, core 0 of cluster 0). However, on the contrary to the isolation mode that shuts down all the other cores of the cluster (thereby nullifying all possible interference within that cluster), we deploy onto all these other cores small programs that we call "Interference Generators" (or IG for short). Those programs are essentially tiny pieces of code that have for sole purpose to saturate all the resources (e.g., interconnection, memory banks) that are shared with the application under analysis running on core 0. Remember that the objective of the contention mode is to create the worst execution conditions for the execution of the analyzed program, i.e. conditions in which its execution is slowed down as much as possible due to contention for shared resources. The IGs are deployed and started *before* the analyzed program starts executing, and they are stopped *after* it has run for a pre-defined number of times.

Implementing the IG that generates the worst possible interference that an application could ever suffer is a very challenging task, if not impossible. This is because the exact behaviour of the application to be interfered with (i.e. its utilization pattern of every shared resources and the exact time-instants of accessing it) should be known, as well as all the detailed specifications of the platform. Besides, even if those information were known, the execution scenario causing the maximum interference may be impossible to reproduce. Rather than concentrating our efforts on creating such a worst IG, we opted for the implementation of an IG that is "bad enough" and used it as a proof of concept to demonstrate how large can be the time-overhead incurred by the application under analysis due to the interference.

Our implementation of the IG consists of a single function "IG_main()" that is executed by a thread dispatched to every core on which the analyzed program is not assigned (recall that the application under analysis is executed sequentially on core 0). That is, every core that is not running the analyzed program runs a thread that executes IG_main(). Essentially, IG_main() executes three functions, namely: IG_init_interference_process(), IG_generate_interference(), and IG_exit_interference_process(). The first one is called upon deployment, at the beginning of execution of IG_main(), before the analyzed program start to execute and be timed. The second one is the main function. It creates interference on the shared resources. The call to that function is encapsulated in a loop that terminates only when the IG is explicitly told to stop. Finally, the third function is called when the analyzed application has been timed and the analysis process is about to end.

Let us now briefly describe our implementation of these three functions on the Kalray MPPA-256. We use a global array of integer called "my_array" and declare the three main functions described above as follows.

```
int* my_array;
inline void IG_init_interference_process() __attribute__((always_inline));
inline void IG_generate_interference() __attribute__((always_inline));
inline void IG_exit_interference_process() __attribute__((always_inline));
```

The first function "IG_init_interference_process()" simply allocates memory space to "my_array" (the size of 1024 integers) and fills that array with arbitrary values. Note that on the Kalray MPPA-256, a thousand integers occupy roughly half of the private data cache of a VLIW core in a compute cluster. The third function "IG_exit_interference_process()" simply frees the memory space held by "my_array". The second function, "IG_generate_interference()", is the main one and a snippet of its code is presented below.

```
inline void IG_generate_interference() {
   __builtin_k1_dinval();
   __builtin_k1_iinval();
   register int *p = my_array;
   volatile register int var_read;
   var_read = __builtin_k1_lwu(p[0]);
   var_read = __builtin_k1_lwu(p[8]);
   var_read = __builtin_k1_lwu(p[16]);
   // ...
   var_read = __builtin_k1_lwu(p[1015]);
   var_read = __builtin_k1_lwu(p[1023]);
}
```

The function starts by invalidating the content of the data and instruction caches. Then, it reads every element of "my_array", starting from the element K=0 and moving on iteratively from element K to element ((K+8) modulo 1024), until K reaches 1023. This way, every element of the array is read exactly once and every two consecutive readings access data that are located exactly $8 \times 4 = 32$ bytes apart in the memory (the size of an integer is standard on the Kalray, i.e. 4 bytes). This is done on purpose knowing that the private data cache line of every VLIW core in the compute clusters of the Kalray MPPA-256 is 32 bytes long. Consequently, every reading causes a cache miss and the value must then be fetched from the 2MB in-cluster shared memory, hence it creates traffic on the shared memory communication channels and potentially interfere with the application being analysed. At runtime, this function is called repeatedly in an infinite while-loop until the IG receives the command to stop (that command is sent at the end of the execution of the program under analysis).

By running the application concurrently with these IGs, every request that it sends to read or write a data in the shared memory is very likely to interfere with a read request from one of the IGs. As reported in Section 4, the variation in the execution time between the isolation and contention modes is substantial.

4 Experimental results

We ran each benchmark application one thousand times, each time over the same input data[3], in isolation and contention modes and both with the HP and LP configurations. Tables 2 and 3 expose the results in the LP and HP configurations, respectively. It is important to stress here that for each program, *we used the same input set for the thousand runs*! We did so in order to focus solely on the variation of execution time due to the interference between concurrently-running processes. The input data set that we used is the one provided "by default" that is available immediately on downloading the source code of the benchmark programs from the TACLe website [5]. From the results we made few interesting observations:

- With the LP configuration (Table 2), for 8 out of 13 tested programs, the execution takes the *exact* same time, to the nearest CPU cycle, when running in isolation one thousand times over the same set of inputs. This is true even for the "mpeg2" program that executes in about 2890 millions of cycles. Its execution time remains constant throughout the 1000 runs. Although this constance is commonly assumed in theoretical works (same

[3] Since the inputs are fixed, the remaining variability in the MIET should be caused by the initial hardware state (like contents of caches, state of the branch predictor, etc.).

Table 2 Results in the LP configuration, in which the instruction and data caches are disabled. The columns "min" and "max" are expressed in millions of cycles; the columns "var" are expressed in % and correspond to $(\max - \min)/\min$; the column "factor" is the ratio between the maxima in contention and isolation, i.e. factor = max (CON) / max (ISO).

	ISO			CON			
Name	**min**	**max**	**var (%)**	**min**	**max**	**var (%)**	**factor**
ammunition	1148.3	1148.3	0	8675.44	8676.04	0.007	7.56
cjpeg_jpeg6b _wrbmp	1.09	1.09	0	8.04	8.05	0.14	7.37
cjpeg_transupp	39.84	39.84	0	279.97	280	0.012	7.03
dijkstra	472.61	472.61	< 0.001	1552.71	1557.35	0.3	3.3
epic	64.15	64.15	0	510.87	511.03	0.031	7.97
gsm_decode	19.69	19.7	< 0.022	151.69	151.84	0.099	7.71
gsm_encode	47.19	47.19	0	340.17	340.22	0.013	7.21
h264_dec	0.46	0.46	0	3.83	3.83	0.126	8.39
mpeg2	2890.55	2890.55	0	20237.81	20238.08	0.002	7
ndes	0.63	0.63	< 0.548	4.46	4.5	0.753	7.09
rijndael_decoder	31.06	31.06	< 0.001	228.46	228.48	0.011	7.36
rijndael_enc	35.16	35.16	< 0.001	256.62	256.73	0.046	7.3
statemate	0.39	0.39	0	3.15	3.15	0.187	8.02

input → same execution path → same output and same duration), we did not expect it to be 100% true in practice.

- With the HP configuration (Table 3), still in the isolation mode, all the programs have experienced a different execution time, which is thus due to the non-determinism of the cache. Sometimes this variation is small, still it is always there.
- Comparing the results of the contention mode between the LP and HP configurations, we see that using the caches has somewhat isolated the programs from each other. Under the HP configuration, The IGs are able to increase the execution time of the analyzed program only by a small factor (1.11 being the worst-case observed). This is because once the program is loaded into the cache (both instructions and data), the program does not need to further communicate with the 2MB of shared in-cluster memory. Therefore the IGs do not have a mean to interfere substantially with its execution. In the LP configuration however, the execution time is increased by a factor up to 8, which means that the analyzed program is 8 times slower due to interference with concurrently-running processes! This slow-down factor clearly advocates the use of specialized techniques to prevent processes from interfering with each other at runtime (or at least mechanisms should be set to mitigate the effect of this interference). We believe that a slow-down factor of similar magnitude could be observed even with the caches enabled (i.e. in the HP configuration) if the analyzed program had to communicate frequently with the shared memory, hence giving the opportunity to the other processes running on the other cores to interfere with its execution.

Important note: All the benchmarks that we have analysed seem to have data sets that mostly fit into the data cache of the core on which they are deployed. It would be useful and highly interesting to conduct further experiments on benchmarks with larger data set sizes. Intuitively, it seems that the results of the "contention" mode for benchmarks with larger data set sizes would be somewhere in between what we see in this paper for the LP and HP

Table 3 Results in the HP configuration, in which the instruction and data caches are enabled. The columns "min" and "max" are expressed in millions of cycles; the columns "var" are expressed in % and correspond to $(\max - \min)/\min$; the column "factor" is the ratio between the maxima in contention and isolation, i.e. factor = max (CON) / max (ISO).

	ISO			CON			
Name	**min**	**max**	**var (%)**	**min**	**max**	**var (%)**	**factor**
ammunition	247.24	247.25	< 0.002	248.07	248.09	< 0.008	1.003
cjpeg_jpeg6b _wrbmp	0.23	0.23	< 0.067	0.23	0.23	< 0.217	1.003
cjpeg_transupp	8.9	8.9	< 0.004	8.9	8.9	< 0.005	1.00003
dijkstra	101.92	101.92	< 0.001	101.39	101.4	< 0.009	0.995
epic	18.81	18.81	< 0.002	18.84	18.84	< 0.023	1.002
gsm_decode	4.44	4.44	< 0.017	4.44	4.45	< 0.042	1.002
gsm_encode	11.08	11.08	< 0.001	11.1	11.1	< 0.028	1.002
h264_dec	0.09	0.09	< 0.149	0.09	0.09	< 0.347	1.002
mpeg2	619.77	619.77	< 0.001	620.31	620.33	< 0.003	1.0009
ndes	0.13	0.13	< 0.679	0.13	0.13	< 0.937	1.003
rijndael_decoder	9.29	9.29	< 0.002	9.54	9.55	< 0.102	1.03
rijndael_enc	10.13	10.14	< 0.105	10.31	10.32	< 0.164	1.02
statemate	0.09	0.09	< 0.33	0.1	0.1	< 1.83	1.11

configurations. Due to time and space constraints, we were not able to include such results in this paper.

5 Conclusion

The paper aimed at quantifying the effect of the inter-process interference on the processor shared resources. We showed that on the Kalray MPPA-256 (Andey) manycore platform the interference can slow down the execution of a program by a factor of 8, and this slow down factor is obtained in conditions that may not even be the worst (the IGs certainly do not generate the maximum interference). Of course, many questions remain open: What is the maximum slow-down factor that we could experience at runtime? What is the relation between the slow down factor and the memory access pattern of the analyzed program? And of course, how to totally isolate the processes from each other without degrading too much the performance? We plan to make more elaborated experiments in the near future to answer those questions, or at least to provide insights that would enable us to answer them.

References

1 Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. Guaranteed services of the noc of a manycore processor. In *Proceedings of the 2014 International Workshop on Network on Chip Architectures, NoCArc'14, Cambridge, United Kingdom, December 13-14, 2014*, pages 11–16, 2014. doi:10.1145/2685342.2685344.

2 Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014. doi:10.7873/DATE.2014.110.

3 Benjamin Lesage, David Griffin, Sebastian Altmeyer, and Robert I. Davis. Static probabilistic timing analysis for multi-path programs. In *2015 IEEE Real-Time Systems Sym-*

posium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015, pages 361–372, 2015. doi:10.1109/RTSS.2015.41.

4 Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. Methodologies for the WCET analysis of parallel applications on many-core architectures. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 748–755, 2015. doi:10.1109/DSD.2015.105.

5 Timing Analysis on Code-Level. http://http://www.tacle.eu/index.php/activities/taclebench. Accessed: 2016-05-06.

6 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem; Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

7 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

BEST: a Binary Executable Slicing Tool

Armel Mangean[1], Jean-Luc Béchennec[2], Mikaël Briday[3], and Sébastien Faucou[4]

1 **École Centrale de Nantes, IRCCyN UMR 6597, Nantes, France**
2 **CNRS, IRCCyN UMR 6597, Nantes, France**
3 **Université de Nantes, IRCCyN UMR 6597, Nantes, France**
4 **Université de Nantes, IRCCyN UMR 6597, Nantes, France**

Abstract

We describe the implementation of Best, a tool for slicing binary code. We aim to integrate this tool in a WCET estimation framework based on model checking. In this approach, program slicing is used to abstract the program model in order to reduce the state space of the system. In this article, we also report on the results of an evaluation of the efficiency of the abstraction technique.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases Program Slicing, Binary Code Analysis, WCET Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2016.7

1 Introduction

In the recent years, several works have explored techniques to statically estimate the worst-case execution times (WCET) of a program using model checking [10, 6, 4]. The most important issue encountered when using model checking to perform WCET estimation is the exponential size of the state space that must be exhaustively explored during the analysis [20]. To fight this problem, state-of-the-art model checking tools for dense timed systems such as Uppaal [14] use powerful symbolic algorithms and data structures. It has been shown that it allows to deal with small but realistic instances of the WCET problem [10, 6]. It is expected that model checking technology will continue to improve in the coming years, widening the range of instances that can be solved.

A different and complementary direction to deal with the explosion of the state space consists in abstracting the models of the programs [4, 3] or the models of the hardware components [5]. The idea is to remove the information which does not impact the WCET. This work follows this direction, with a focus on the models of programs. In the continuation of prior work [4] we explore the use of program slicing [19] at the level of the binary code to abstract the model of the program.

In this paper we introduce Best, a program slicer for binary code. We describe its architecture and implementation. We explain the interface between Best and Harmless [12], a toolchain built around a Hardware Architecture Description Language (HADL). Thanks to this interface, the core of Best is independent from the target instruction set of the binary code. We also use Best and the Mälardalen benchmarks to show how to compute abstract model of programs and report on the benefits that could be reached with this approach.

The paper is organized as follows. In Section 2 we give an overview of related works. In Section 3 we outline an approach to the estimation of WCET with model checking. In Section 4 we provide a summary of program slicing. In Section 5 we describe the

implementation of Best and its interface with Harmless. In Section 6 we report on an evaluation of the abstraction approach using Best. In Section 7 we conclude the paper.

2 Related works and contribution

In the context of static WCET analysis, program slicing has been explored [17, 15, 4]. Program slicing is mostly used to accelerate the static analysis of flow facts [17, 15]. Our goals are different, as well as the slicing technique. In contrast to our work, program slicing is applied to structured programs at the source code level (or intermediate code level [17]). Our tool works at the binary code level. As a positive side effect, it is independent from both the programming language and the compiler. To our best knowledge, there is no established tool for slicing non-x86 binary code and Best aims at filling this gap.

Our work is the continuation of previous work by Cassez and Béchennec [4]. In this work they propose a prototype tool based on the classical dataflow equations approach [19] that computes slices for ARM-based binary code. Unlike that, our tool is independent from the target instruction set thanks to its interface with the Harmless toolchain. Furthermore, our tool is based on a state-of-the-art graph-based approach [13]. We also provide an evaluation focused on the benefits of the program abstraction technique.

Brandner and Jordan [3] propose a graph pruning technique to increase the precision of static WCET estimation. Branches of the Control Flow Graph (CFG) are pruned based on the criticality of their basic blocks. The criticality is defined as the normalized duration of the longest path passing through the block [2]. According to the authors this approach is akin to "program slicing in the time domain". Based on this pruning approach, a refinement based WCET calculation meta-algorithm is proposed. We do not address a full WCET analysis in this paper. However, their technique could be combined with our approach to improve WCET calculation. Such a combination should allow to further abstract the program in order to deal with state space explosion.

3 WCET estimation using model-checking

WCET estimation can be reduced to a reachability problem in a network of timed automata [4]. The Uppaal tool that supports timed automata extended with bounded integer variables is used to build the models, and to solve the reachability problem.

A model of the hardware is built where each architectural feature (pipeline(s), cache(s), bus(es), memory, ...) is modeled by one or more timed automata. These automata are synchronized through *channels* to model the actual hardware behavior. For instance the automaton modeling the fetch stage of a pipeline is synchronized with the automaton modeling the instruction cache which is synchronized with the automaton modeling the bus and so on. The timings of the hardware are modeled by guards and clocks on some edges of the automata. A simple model of a memory controller could be the timed automaton of Figure 1. Notice that this model accounts only for the timing.

A model of the program is automatically built from the binary code. In this model, each location corresponds to an instruction. An edge leaving a location corresponds to the execution of the instruction. For conditional branches, two edges leave the location according to the behavior of the branch (taken or not taken). Each edge is synchronized with the automaton that models the instruction fetch so that it may only be fired if the hardware fetches a new instruction. Memory locations are updated according to the semantics of the instruction and to its advance in the pipeline. The model of the program has an initial state,

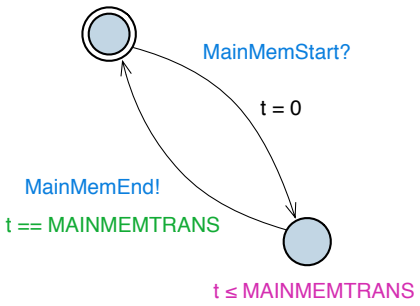


Figure 1 Simple modeling of a memory using UPPAAL. In the initial state (on the top left) the memory waits for an access (MainMemStart synchronization channel). When the access is requested, clock t resets and the automaton remains in the bottom right state until t reaches the MAINMEMTRANS value. Then the memory returns to the initial state and notifies the end of the memory access (MainMemEnd synchronization channel).

I, that corresponds to the entry point of the program and a final state, F, that corresponds to the point at which the WCET has to be computed.

At last, a global clock x is used to measure the time. It is initialized at 0.

The WCET is then the largest value, $max(x)$, of x when F is reached. $max(x)$ can be computed with a model-checker and the following reachability property $R(T)$: "Is F reachable with $x \geq T$?". If $R(T)$ is true and $R(T+1)$ is false then T is the WCET of the program.

This approach is modular since the hardware and software models are built separately and the hardware model does not depend on the software to check. No assumption is made about the structure of the binary code generated by the compiler and the model of the program is built automatically without need for annotations

Modeling the values stored in memory

Data stored in memory and registers – called a *location* in the remaining of the paper – and used by the program can be either included in or abstracted away from the model. Each location included in the model is associated with a bounded variable. When the program accesses a location, the timing is computed by the models of the hardware. If the location is included in the model, the associated variable is also read / written. If the location is abstracted away, the data to be written is discarded and any read access returns the special value $\perp$.

On the one hand, every location included in the model adds a dimension to the state of the system and thus contributes to the growth of the state space. On the other hand, $\perp$ values can lead the model checker to explore paths that are not in the systems when they impact a conditional branch instruction. Thus the problem is to automatically compute the minimal set of locations that impact on the control flow of the program and that should be included in the model. In this paper, we focus on this problem.

4 Program Slicing

4.1 Notations

Let $\mathcal{I}$ be a finite set of instructions. Let $\mathcal{L}$ be a totally ordered finite set of labels. A program P is a finite subset of $\mathcal{L} \times \mathcal{I}$ such as $\forall (l,i) \in P, \quad (l,i') \in P \leftrightarrow i = i'$. We denote $\mathcal{V}$ the set

of variables of P. If we consider the program in Figure 2a, $\mathcal{I}$ is the subset of instructions of the 32 bits PowerPC instruction set used by the program, $\mathcal{L}$ is the set of memory addresses aligned on 4 bytes boundaries in the range $[3000, 3034]$ and $\mathcal{V}$ is the set of memory locations explicitly or implicitly used (i.e. $\{r1, r3, r8, r9, r10, lr, ctr\}$).

A basic block is a sequence of instructions of P with one entry point, its first instruction, and one exit point, its last instruction. A basic block is maximal if it is not contained in any other basic block. Let $G_P = \langle V_P, E_P, u_{G_P}, v_{G_P} \rangle$ where V_P is the finite set of maximal basic blocks of P and $E_P \subset V_P \times V_P$ is such that there is an edge between $v_1 \in V_P$ and $v_2 \in V_P$ if and only if the first instruction of v_2 can be executed immediately after the last instruction of v_1 in P. $u_{G_P} \in V_P$ and $v_{G_P} \in V_P$ are respectively the entry block and the exit block of P. Then G_P is the CFG of P.

4.2 General overview

Program slicing has been introduced by Weiser [19]. Weiser defines a program slice as an executable program that is obtained from the original program by deleting zero or more statements, computing the same values for a given subset of variables of the program. He claims that a slice corresponds to the mental abstractions that people make when they are debugging a program. The original formulation of program slicing proposed by Weiser is based on iterative solutions of data-flow equations. Ottenstein and Ottenstein [16] were the first to redefine slicing as a reachability problem in a dependence graph representation of a program. They use a Program Dependence Graph (PDG) [8] for static slicing of single-procedure structured programs. Efforts have been made to extend this approach to unstructured programs [1, 13] and multiple-procedure programs [11, 13]. More details on the topic can be found on the survey by Tip [18].

We consider in this section a toy example to highlight the slicing method. It is a simple program that computes iteratively the first 30 values of the Fibonacci sequence ($F_n = F_{n-1} + F_{n-2}$, with $F_0 = 1$ and $F_1 = 1$). The code targets the PowerPC instruction set. The program works as follow:

- The `_start` label (Figure 2a, line 1) is the program entry point. It gets minimal startup code that initializes the stack pointer $r1$ and calls the `main` at 3010 (Figure 2a, line 7). If the `main()` function returns, it enters in an infinite loop (Figure 2a, line 5) ;
- Figure 2a, lines 8 to 11 initialize the sequence. The loop is controlled by the dedicated *ctr* counter register ;
- Figure 2a, lines 13 to 16 are the instructions in the loop. $r9$ and $r10$ stores respectively the current and the last value and are used to compute the next value (in $r3$).

A slice is computed with regards to a slice criterion $\mathcal{C} = \langle l, v \rangle$ with $l \in \mathcal{L}$ a label and $v \subseteq \mathcal{V}$ a set of variables. So, if we consider the program in Figure 2a and the slicing criterion $\langle 3030, \{ctr\} \rangle$, i.e. the value of register *ctr* when the instruction pointer contains the address 3030, we obtain the slice shown in Figure 2b. Indeed, the instruction `bdnz 3024` at address 3030 (Figure 2b, line 16) implicitly modifies the register *ctr*, *ctr* is set by `mtctr r8` at 3018 (Figure 2b, line 10) and $r8$ is set by `li r8,29` at 3010 (Figure 2b, line 8).

To compute a slice in binary code, we need to handle arbitrary control flows (as opposed to control flow of structured programs) and inter-procedurality. In our use case, we must also exclude the techniques that change the order of the instructions. Given all these constraints, we have to use slicing techniques based on graph manipulations [13].

This approach is based on the computation of several graphs. The first one is the CFG of the program. Figure 3a gives the CFG of `fibcall-O2.elf`. Then the Data Dependence

```
1  00003000 <_start>:
2       3000:   li     r1,1            ;r1  <- 1
3       3004:   ori    r1,r1,49296     ;ri
   <- r1 | 49296
4       3008:   bl     3010            ;call main
5  0000300c <loop>:
6       300c:   b      300c            ;branch
7  00003010 <main>:
8       3010:   li     r8,29           ;r8  <- 29
9       3014:   li     r10,1           ;r10 <- 1
10      3018:   mtctr  r8              ;ctr <- r8
11      301c:   li     r9,1            ;r9  <- 1
12      3020:   b      3028            ;branch
13      3024:   mr     r9,r3           ;r9  <- r3
14      3028:   add    r3,r9,r10       ;r3
   <- r9+r10
15      302c:   mr     r10,r9          ;r10 <- r9
16      3030:   bdnz   3024            ;ctr--,
17                                     ;branch if ctr!=0
18      3034:   blr                    ;return
```

(a) Dump of `fibcall-O2.elf`.

```
1  00003000 <_start>:
2       3000:   --     --
3       3004:   --     --
4       3008:   --     --
5  0000300c <loop>:
6       300c:   --     --
7  00003010 <main>:
8       3010:   li
   r8,29
9       3014:   --     --
10      3018:   mtctr  r8
11      301c:   --     --
12      3020:   --     --
13      3024:   --     --
14      3028:   --     --
15      302c:   --     --
16      3030:   bdnz
   3024
17
18      3034:   --
```

(b) Slice for $\mathcal{C} = \langle 3030, \{ctr\}\rangle$.

Figure 2 Dump and slice of a binary executable.

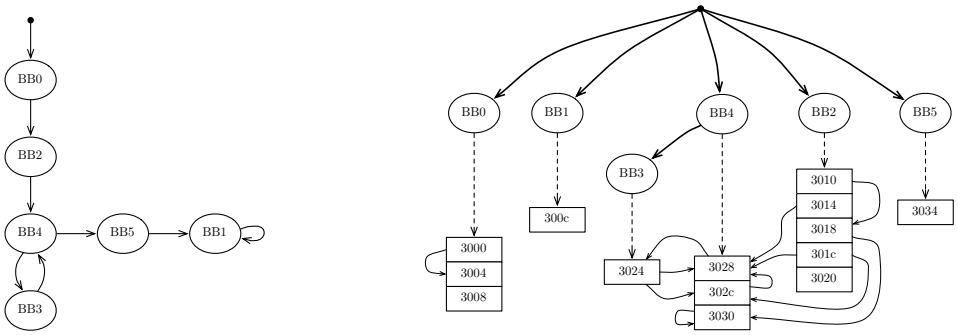
Graph (DDG) and the Control Dependence Graph (CDG) are computed from the CFG. The DDG captures data dependencies between instructions. Its nodes are the instructions of P. There exists an edge between two nodes of the DDG when the source node does a reaching definition of a memory location used by the target node. The CDG captures control dependencies between basic blocks. Its nodes are the maximal basic blocks of P. There exists an edge between two nodes of the CDG when the source node determines whether the target node is executed or not.

After the DDG and the CDG, the next graph is the Program Dependence Graph (PDG) [8]. It is built by merging the DDG and the CDG. Node sets of the DDG and the CDG being disjoint (nodes are instructions in the DDG and maximal basic blocks in the CDG), the PDG gets its consistency from special edges that represent the belonging of a set of instructions to a basic block. In summary, the PDG captures the belonging of set of instructions to basic blocks, data dependencies at instruction level and control dependencies at basic block level. Figure 3b gives the PDG of `fibcall-O2.elf`.

If P does not contain procedure calls, or if these calls are "inlined" when the CFG is built, it is possible to compute slices on the PDG. The slice corresponding to a given criterion is obtained by performing a backward reachability analysis. The slice is initialized with the slice criterion. When an instruction in the slice is the target of a data dependence edge, the source instruction is added to the slice. When an instruction in the slice belongs to a basic block which is the target of a control dependence edge, the last instruction of the source basic block is added to the slice. This procedure is iterated until a fixpoint is reached.

In Figure 3b, dashed, bold and solid edges represent respectively the belonging of a set of instructions to a block, a control dependency between two basic blocks, and a data dependency between two instructions. Considering once again the program in Figure 2a and the slicing criterion $\langle 3030, \{ctr\}\rangle$, we obtain the slice shown in Figure 2b. Indeed, the backward reachability analysis shows that the instruction at address 3030 has a data dependency with the instruction at 3018 which has also a data dependency with the instruction at 3010 and the basic block $BB2$ has no control dependency apart from the entry point.

Slicing the PDG is suboptimal for programs with procedure calls [13]. To overcome this limitation, inter-procedural slicing techniques use a fourth graph, the System Dependence Graph (SDG) [11]. To build the SDG, in a first step, the PDG of each procedure must be built. In a second step, these PDGs are connected with call, parameter-in and parameter-out edges to account for procedure calls and parameters passing. The slicing algorithm on the SDG is based on two backwards analyses similar to the one used for the PDG. The first



(a) CFG of `fibcall-O2.elf`. **(b)** Simplified PDG of `fibcall-O2.elf`.

Figure 3 Dump and slice of a binary executable.

backward analysis does not follow parameters-out edges. It only adds to the slice instructions up to the entry point. The second backward analysis does not follow call and parameter-in edges. It adds to the slice all instructions down to the procedures output parameters. As a result, unwanted dependencies to output parameters from called procedures are not added to the slice.

4.3 Abstraction of programs for WCET estimation.

Program slicing has many use cases in software engineering. In this paper we want to compute the set of memory locations that impact the WCET of a program. To determine this set of locations we have to determine a suitable slicing criterion. This criterion is the set of pairs $\langle l, v \rangle$ such that l is the label of a conditional branch instruction and v is the set of memory locations read by this instruction. If we consider the program in Figure 2a, it has only one conditional branch instruction: `bdnz 3024` at address 3030. The branch is taken if the count register *ctr* is not zero. So, to compute the locations that should be part of the state of the model we have to compute the slice for the criteria $\{\langle 3030, \{ctr\}\rangle\}$. The set of variables used either explicitly or implicitly by the initial program is $\{r1, r3, r8, r9, r10, lr, ctr\}$. The subset of variables used in the slice is $\{r8, ctr\}$ (see Figure 2b). Only these two registers have to be included in the state of the model.

Let us underline that computing this slice gives us extra informations. For each register in the slice, we also know which instructions impact its value at a given execution point. In the general case, not all the instructions using a register in the slice are in the slice. Such instructions must be processed as instructions using registers not in the slice. Their output must not be written to the state. This allows to further reduce the number of states to explore.

5 Implementation

Architecture

Our tool, BEST for Binary Executable Slicing Tool, computes slices on binary executable files. Its architecture is illustrated in Figure 4.

The decoding and interpretation of the binary files relies on a library generated by the HARMLESS toolchain [12]. HARMLESS is an Hardware Architecture Description Language that is used to model a whole processor. In this study, we are only interested in the model

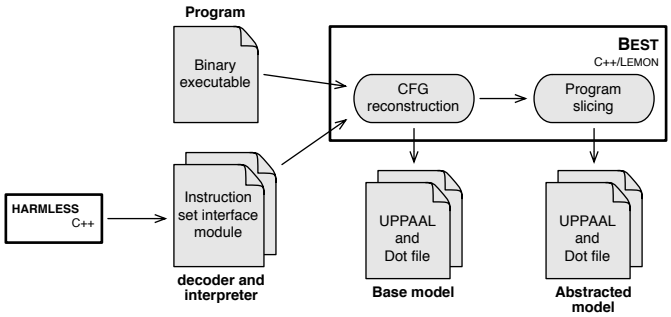


Figure 4 Structure of the tool.

of the instruction set. The HARMLESS compiler is primarily designed to generate either functional or cycle accurate simulators. We re-targeted it to extract static information of the instruction set. The library generated from HARMLESS can read a binary file (`.elf` format in our case) and give information about each instruction such as:

- the instruction mnemonic;
- the memory locations that are read by the instruction;
- the memory locations that are written by the instruction;
- is the instruction a branch instruction? is it a conditional branch? what is its target (if it is statically defined)?

In this study we have used only the PowerPC instruction set, but BEST is not architecture-dependent, thanks to this library.

Using this library, BEST does a CFG reconstruction from a PowerPC binary executable file. Then it applies program slicing to compute the set of memory locations that should be in the model. The main output is an abstract model of the program that can be used to solve the WCET estimation problem with UPPAAL. For validation and visualization purposes, the different models built along the computation can be exported as graphs or as timed automata in the UPPAAL format [14].

BEST is distributed in open-source[1]. To the best of our knowledge, there is no established program slicing tool for non-x86 binary code, especially in open-source. BEST aims to fill this gap. It is implemented in C++. Apart from HARMLESS it relies on the graph manipulation library LEMON [7].

Limitations and Future work

The current version of BEST has different limitations that we want to break in the near future. First, the computation of the abstraction is limited to the register file. The other levels of memory (stack words, all other parts of the volatile memory and non-volatile memory) are automatically excluded from the model. It will be straightforward to take into account the other levels by extending the technique used for the register file. The first step will be the analysis of the stack frame. Being able to track data dependencies between memory and registers through stack loads and stores will produce a more accurate model of the binary executable, and so more accurate WCET estimations.

The second limitation is the limited support for programs with multiple procedures. The slice is currently computed on the PDG. It is not much of hard work to build the SDG and

[1] Available at `https://github.com/TrampolineRTOS/BEST`.

Table 1 Ratio of registers (resp. instructions) in slice compared to the unsliced program.

Compiler	Optim.	Registers in slice				Instructions in slice			
		Avg.	Min	Max.	Std. dev.	Avg.	Min	Max.	Std. dev.
Gcc	`-O0`	61.8%	43.7%	78.9%	8.8%	21.6%	1.7%	43.2%	8.9%
	`-O1`	64.6%	19.1%	87.5%	13.6%	36.7%	3.24%	67.9%	10.2%
	`-O2`	64.3%	13.3%	92.9%	20.3%	37.8%	2.9%	72.5%	15.7%
	`-O3`	62.8%	9%	96.4%	21.7%	34.7%	0.4%	72.2%	18%
Cosmic	`-no`	40.5%	8.6%	86.7%	16.6%	34%	1.9%	60.2%	15.2%
	default	37%	2.8%	66.7%	15.8%	37%	2.8%	66.7%	15.8%

adapt the slicing algorithm because Best has been designed on structures and algorithms intended to produce inter-procedural slices. The main benefit of inter-procedural slicing resides on a more accurate slicing of procedure parameters i.e. even smaller slices.

6 Experimental results

We have conducted experiments to measure the reduction of the set of memory locations that must be included in the model. Given the current restriction of Best, we have focused on the registers. To do so, Best outputs the following information for each program:

- the number of registers used either explicitly or implicitly and the number of instructions in the original program ;
- the number of registers used either explicitly or implicitly and the number of instructions in the sliced program (using the slicing criterion defined in Section 4).

We used the Mälardalen WCET benchmarks [9] to generate the programs. We had to exclude certain programs to account for the current limitation of our tool: program containing floating point arithmetic or switch-case statements and recursive programs.

We used the library generated by the Harmless compiler from a description of a PowerPC e200z4 core based on the 32 bits PowerPC instruction set. This architecture includes 32 general purpose registers ($r0$, $r1$, ..., $r31$) and 5 dedicated registers (cr, xer, lr, ctr, pc). We used two different compilers: Gcc 5.3.1 and Cosmic C 4.3.7. For a given compiler, the generated binary may be very different according to the optimizations. For instance without optimization Gcc generates code where local variables are loaded from and stored to the stack frame each time they are used, whereas in higher optimization levels local variables are allocated in registers. Thus we created different program versions for each optimization level offered by each compiler (4 levels for Gcc and 2 levels for Cosmic C).

All in all, we created 6 versions of each of the 16 Mälardalen benchmarks fitting our constraints and we ran Best on these 96 programs. Due to space limitations the detailed results are provided online[2]. The results are summarized in Table 1 and 2. Table 1 gives the ratio of registers and instructions in the slice compared to the original program. Table 2 gives the number of registers in the slice. It is not meaningful to compare our results with pior work [4] because we consider a different instruction set and different compilers (or at least compiler version for Gcc). We do not comment either on the execution time of Best that were below one second in every case.

[2] Available at `https://github.com/TrampolineRTOS/BEST`.

Table 2 Average number of registers in slice.

Gcc				Cosmic	
`-O0`	`-O1`	`-O2`	`-O3`	`-no`	*default*
8.8	13	11.8	12.1	11.9	12

These results confirm that slicing is an effective abstraction technique for our use case. It allows a significant reduction of the number of variables that should be included in the model (reduction of the dimension of the state space) as well as the number of instructions the output of which should be taken into account (reduction of the number of states to explore). As expected, the best results are obtained for programs with very simple control flow, namely `fdct.c` and `jfdctint.c`, whereas the worst results are obtained for programs with nested control statements and procedure calls , namely `ndes.c` and `adpcm.c`. However, let us underline that the structure of the source code is not always the dominant factor. For some programs, it appears that the compiler (version and/or optimization) has more impact on the capacity of the program slicer to abstract the binary. Example of such programs are `expint.c` and `fir.c`.

7 Conclusion

This article describes the working principles of a tool that computes abstract models of binary executables to be processed by a WCET estimation toolchain based on model checking. Our tool uses program slicing to compute the set of memory locations of the program that have an impact on the WCET of the program. The content of these memory locations is tracked in the abstract model of the program whereas the content of the other ones is abstracted away. A first prototype has been implemented and evaluated. The evaluation has been performed on the Mälardalen benchmarks using two compilers for the PowerPC architecture with varying optimization levels. On average, 41% of the registers can be abstracted. This is a promising result.

References

1 Hiralal Agrawal. On slicing programs with jump statements. *ACM Sigplan Notices*, 29(6):302–312, 1994.

2 Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In *International Conference on Real-Time and Network Systems (RTNS)*, 2012. doi:10.1145/2392987.2393000.

3 Florian Brandner and Alexander Jordan. Refinement of worst-case execution time bounds by graph pruning. *Computer Languages, Systems & Structures*, 40(3-4):155–170, 2014. doi:10.1016/j.cl.2014.09.001.

4 Franck Cassez and Jean-Luc Béchennec. Timing Analysis of Binary Programs with UPPAAL. In *International Conference on Application of Concurrency to System Design (ACSD)*, 2013.

5 Franck Cassez and Pablo González de Aledo Marugán. Timed automata for modeling caches and pipelines. In *Workshop on Models for Formal Analysis of Real Systems (MARS)*, 2015. doi:10.4204/EPTCS.196.4.

6 Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: modular execution time analysis using model checking. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET*

2010, July 6, 2010, Brussels, Belgium, pages 113–123, 2010. doi:10.4230/OASIcs.WCET.2010.113.

7 Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON – an Open Source C++ Graph Template Library. *ENTCS*, 264(5):23–45, 2011.

8 Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 136–146, 2010. doi:10.4230/OASIcs.WCET.2010.136.

10 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112, 2010. doi:10.4230/OASIcs.WCET.2010.101.

11 Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

12 Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Guillaume Savaton, and Yvon Trinquet. Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. *JSA*, 58(8):318–337, 2012.

13 Akos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural Static Slicing of Binary Executables. In *International Workshop on Source Code Analysis and Manipulation*, 2003.

14 Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.

15 Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *International Symposium on Code Generation and Optimization (CGO)*, pages 136–146, 2009.

16 Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

17 Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 103–112, 2006. doi:10.1145/1134650.1134666.

18 Frank Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3(3), 1995.

19 Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

20 Reinhard Wilhelm. Why AI + ILP is Good for WCET, but MC is not, nor ILP alone. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2004.

Dynamic Branch Resolution Based on Combined Static Analyses*

Wei-Tsun Sun[1] and Hugues Cassé[2]

1 **IRIT, University of Toulouse, Toulouse, France**
`wsun@irit.fr`
2 **IRIT, University of Toulouse, Toulouse, France**
`casse@irit.fr`

Abstract

Static analysis requires the full knowledge of the overall program structure. The structure of a program can be represented by a Control Flow Graph (CFG) where vertices are basic blocks (BB) and edges represent the control flow between the BB. To construct a full CFG, all the BB as well as all of their possible targets addresses must be found. In this paper, we present a method to resolve *dynamic* branches, that identifies the target addresses of BB created due to the switch-cases and calls on function pointers. We also implemented a slicing method to speed up the overall analysis which makes our approach applicable on large and realistic real-time programs.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases WCET, static analysis, dynamic branch, assembly, machine language

Digital Object Identifier 10.4230/OASIcs.WCET.2016.8

1 Introduction

The verification of critical real-time system is utterly important to ensure safety and to avoid catastrophic failures. An aspect of this verification is related to check the schedulability of the real-time programs, which requires computing the Worst Case Execution Time (WCET). To be precise, the WCET computation needs to be performed at machine code level to cope with all details of the work of the underlying microprocessor and memory system.

Implicit Path Enumeration Technique (IPET), one of the most effective approach to compute the WCET by static analysis, consists in three phases: (a) the execution path analysis, (b) the block timing analysis, and (c) the WCET estimation as a maximisation of an object function modelled as an Integer Linear Programming system (ILP). Usually the execution paths are represented synthetically by a Control Flow Graph (CFG) where vertices are basic blocks (BB)[1] and edges represent the control flow between the BB. This phase is crucial because, if some paths are missing, the obtained WCET will not be sound.

Yet, working at the machine-code level means that the analyser has to retrieve the control flow from semantically-poor instructions, as the results of translating the rich high-level language structures and of the optimisations performed by the compiler to speed up the program. The CFG may be viewed as a kind of canonical representation of the execution path independent of the high-level language. Yet, some paths are harder to recover. Usually,

* This work is supported by the french research foundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001).

[1] A BB is a sequence of instructions which are only started at the first instruction and which accepts only the last instruction as a control instruction.

the execution paths are determined by executing control instructions (such as branches) that can be executed conditionally or unconditionally. Different execution paths are the result from control instructions that modify the program counter of the microprocessor to execute one part of the code (e.g. the branch is taken) or another. Most of these instructions are *static*: the target address is obtained as a combination of the instruction program counter and of the literal operand (e.g. a constant) found in the instruction word (therefore known at analysis time). Such control instructions are used to translate selection or loop structures.

In the opposite, the target addresses of some high-level structures are results of complex computations based (a) on the current state of the program or/and (b) on the data stored in memory. For example, the C language supports the concept of function pointer meaning that the control flow depends on the data flow of the program and the program points that set this pointer. Another construct of the C language, the `switch`-case statement, may be translated and optimized as a branch whose possible target addresses are stored in a look-up table: an index is computed from the case value and used to get the corresponding index of the table. We call this type of control instruction, *dynamic*: to be analysed, they require data-flow information that is usually obtained, in turn, from an analysis of the CFG.

The contributions and the organisation of the paper

This paper proposes a new approach to determine the possible target(s) of *dynamic* control-flow instructions based on the combination of different types of analyses, the Circular-Linear Progression (CLP) [5, 11] and the k-set analyses. We are also introducing *LightSlicing*, a policy of program slicing which does not require *address analysis* for both simplicity and better performances. LightSlicing works on the machine code and (1) is relatively cheap in computation time and (2) remains precise enough to slice out the program parts that are not involved in the calculation of the control flow targets. Hence, the reduction of the analysis time for both small benchmarks and large realistic programs. We believe our solution is well-adapted to industrial applications.

The remaining of the article is organized as follows: in Section 2 we look into the problem of *dynamic* branches and expose our combined analyses approach. Section 3 shows our approach for the fast program slicing to speed-up the analysis. The experimentation and the related works are presented, respectively, in Section 4 and 5. Finally, Section 6 concludes the paper and also proposes possible extensions of our approach.

2 Dynamic Branch Resolution

This section presents the combined analysis used to resolve the targets of *dynamic* branch.

2.1 Path Analysis

The path analysis aims to provide a representation of the execution paths of the program. In this paper, we focus on the CFG representation, a graph $G = (V, E, \epsilon)$ where V is the set of BB, the set $E = V \times V$ is the set of edges, and the vertex $\epsilon \in V$ is the entry of the graph. G is built from the binary representation of the program by following the instruction flow from known entry points that may be the starting point of the program or function entries provided in the symbol table of the binary file. Usually, it is not feasible to sequentially decode the code segments because (a) they may also contains data and (b) some instruction sets have variable-size instruction words.

Algorithm 1 CFG Building.

```
1:  V ← {ϵ}; E ← ∅; wl ← {(ϵ, i_0)}
2:  while wl ≠ ∅ do
3:      (v, i) ← pop(wl)
4:      B ← [i]
5:      while ¬ is_control(i) do      ▷ if the current instruction is not a control instruction
6:          i ← next(i); B ← B @ [i]      ▷ append the instruction to the BB
7:      end while
8:      if B ∉ V then
9:          V ← V ∪ {B}      ▷ collect the current BB
10:     end if
11:     E ← E ∪ {(v, B)}; wl ← wl ∪ {(B, target(i))}      ▷ creating the edges between BBs
12:     if is_conditional(i) then      ▷ for conditional instructions, such as BEQ
13:         wl ← wl ∪ {(B, next(i)}      ▷ the next instruction will be used to start a new BB
14:     end if
15: end while
```

Algorithm 1 gives an overview of how to build a CFG. In brief, the sets V and E are built incrementally from the synthetic initial BB, ϵ and from the initial instruction i_0. A BB is obtained from continuously collecting the current instruction i until reaching a control instruction. Depending on the nature of this instruction (conditional or not), the next instruction and/or the target instruction of the branch are added to the working list wl. When wl is empty, all paths have been traversed and the CFG is complete.

Functions *is_control*, *is_conditional*, and *next* are instruction-set dependent but can be easily derived from the instruction words, which also applies for the *target*s of *static branches*. However, the *target* may also be resulted from a computation involving the state of the program: such an instruction is called a *dynamic branch*. Having dynamic branches leads to an incomplete CFG. To compute its possible targets, an analysis of the possible program state is required.

2.2 The Flow of the Dynamic Branching Analysis

Let's take the example of Fig 1 that implements a `switch`-case statement. The actual computation of the branch address is performed by instructions at addresses 0x2e260 to 0x2e268. r_3 is loaded from a byte in memory; if it is not greater than 3 (comparison at 0x2e264), it is used to compute the address $pc + r3 \times 4$ that points to an entry of the subsequent table that contains the actual targets of the branch. Hence, (a) the calculation of the possible targets of *dynamic* control instruction 0x2e268 is feasible and (b) we need to use a value analysis to evaluate its components. Another outcome of this example is that we need to apply value analyses to partial CFG and we need also to repeat the analysis until we get the whole CFG: on the first path, the CFG until the *dynamic* control instruction is obtained and values for pc and $r3$ are estimated and enable the calculation of *dynamic* control instruction targets; in the second path, the CFG is extended and, possibly, new *dynamic* control instruction are discovered and so on.

In our approach, the flow to identify the targets of dynamic branches consists of the following steps: (1) First we perform a CLP analysis [5, 11] to obtain the possible values of the registers and the memory addresses. The range of the represented values is over-approximated, i.e. this includes both of all possible values (values which may present during

```
    Addrs   Content    Assembly (ARM)
 1  2e250   e59f2050   ldr    r2, [pc, #80] ; load for r2
 2  2e254   e51b3008   ldr    r3, [fp, #-8] ; load for r3
 3  2e258   e0823003   add    r2, r3, r2 ; r2 used by some cases
 4  2e25c   e59f3264   ldr    r3, [pc, #612]
 5  2e260   e5d33000   ldrb   r3, [r3]
 6  2e264   e3530003   cmp    r3, #3
 7  2e268   979ff103   ldrls  pc, [pc, r3, lsl #2]
 8  2e26c   ea000075   b      2e448 ; address of the default case
 9  2e270   0002e280   ; target address for the 1st case
10  2e274   0002e2dc   ; target address for the 2nd case
11  2e278   0002e364   ; target address for the 3rd case
12  2e27c   0002e3c4   ; target address for the 4th case
13  2e280   e59f3254   ldr    r3, [pc, #596] ; the first case
```

Figure 1 Example of the switch code in ARM's assembly.

the program execution) and spurious values (which are included due to the analysis because of the performed abstraction). (2) A k-set analysis is then applied to gather the concrete values of the registers and the memory addresses. (3) The dynamic branch resolution is carried out to find the targets of the control instructions. If new targets of a control instruction are found, the CFG of the program will be updated with newly added code segments and the analysis will restart from the step (1). The analysis terminates once reached a fix-point such that no more new targets are added.

2.3 CLP analysis and its drawbacks

The CLP analysis makes use of abstract interpretation [6] with the trade-off of (1) having the better performance, especially when performing analysis on loops, and (2) the accuracies of the analysis, for example the strategy for performing widening.

A range of integers can be represented in the format of CLP, and we call the represented range a CLP value. To differentiate, we use *sub-values* to call the integers within a CLP value. Each CLP value is a triple (b, δ, m) representing set $\{b + \delta i \mid 0 \leq i \leq m\}$: b is the starting integer, δ the amount of difference between integers and m the number of integers within a CLP representation. For example, to represent a set of integers 2, 4, 10 in CLP, we will have base = 2, delta = 2, and multiple = 5, such that the CLP value will cover the set 2, 4, 6, 8, 10. Therefore, one may consider that in the domain of CLP, the set of sub-values is presented in an *over-sampling* manner, i.e. in order to include all the possible (in this case 2, 4, and 10) values, some spurious values (6 and 8 here) are included.

The over-sampling behaviour of CLP is for the sake of soundness but this can also bring unwanted behaviours in the analysis. We use Figure 1 to demonstrate this. Figure 1 contains the ARM instructions typically generated from a switch case and perform as the follows: (1) storing the result of some calculation to the register r_2 (lines 1 to 3), which will be used later; (2) lines 4 and 5 provides the switch-index number used to calculate the target address of the switch-cases; (3) the values of the switch-index, infers as the number of the possible targets, are limited by line 6 so that line 7 will only execute if the switch-index falls in the desired range; and (4) if none of the case is chosen, the default case falls through (line 8). By looking at the lines 6 and 7, we know that the index (stored in r_3) falls between the range 0 to 3. The target addresses to load is calculated as $pc + r_3 \times 4$, which are stored between the

address 0x2e270 to 0x2e27c (lines 9 to 12). The CLP representation of these target addresses is thereby of base = 0x2e280, delta = 0x4, and multiple = 0x51. This indicates that there could be 82 (number of the multiple plus one) potential targets which is a huge difference from the actual amount of the possible targets (which is 4).

2.4 k-set analysis

To overcome the drawback of CLP abstraction, we use a k-set analysis [4]. In contrast with the CLP, the values in the k-set analysis are in the form of sets which size is bound to k values. If we get a set bigger than k, it is approximated to $\top$ (any possible value): this property avoids too long or endless analysis looping to reach a fix-point. The k-set analysis is only slightly better than a constant propagation because it usually does not cope well with most of variable behaviour (often linear): the analysis time would become excessively large. Yet, the branch target addresses are not linear and to avoid the over-sampling problem, they need to be stored as an explicit set and k-set is a good candidate to represent them.

Let $\widehat{S} \subseteq 2^{\mathbb{N}}$ to be the set of k-set values that abstracts concrete value as set over $\mathbb{N}$. The abstraction, $\alpha : \mathbb{N} \to \widehat{S}$ is quite simple: $\forall n \in \mathbb{N}, \alpha(n) = \{\ n\ \}$. It is easy to extend a function $f : \mathbb{N} \to \mathbb{N}$ to $\widehat{f} : \widehat{S} \to \widehat{S}$ by just applying the concrete function f to each element of the input set:

$$\forall\ a \in \widehat{S},\ \widehat{f}(a) = \{\ f(e) \mid e\ \in\ a\} \tag{1}$$

Likely, an abstraction of functions with an arity bigger than 1 may be built by a Cartesian product. Yet, if the size of the resulting set is bigger than k, the resulting value is approximated to $\top$ (the abstraction of any possible value). In the static analysis, in order to reduce the number of paths to explore, we often use a join operator $\sqcup$ to combine together values when several paths of the CFG join. Its definition for k-set is given in Eq. 2: the set union $\cup$ is mainly used while the resulting set size is lower or equal to k. Otherwise, the result is $\top$.

$$\forall\ a,\ b \in\ S,\ a\ \sqcup\ b = \begin{cases} \top & if\ \ |a\ \cup\ b| > k \\ a\ \cup\ b & else \end{cases} \tag{2}$$

$$\forall\ a,\ b \in\ S,\ a\ \nabla\ b = \begin{cases} a & if\ a = b \\ \top & else \end{cases} \tag{3}$$

Finally, to speed up the convergence of paths containing loops, a widening operator $\nabla : \widehat{S} \times \widehat{S} \to \widehat{S}$ is useful. As the usual k-set implementation exhibits very poor performances, because of the number of generated values in a loop context, we use a stringent implementation of ∇ in Eq. 3: if both operands are the same the result is this value, else the $\top$ value is returned. This definition works well with purpose of our k-set analysis: the code addresses are rarely, maybe never, the result of a computation and even less the result of a loop computation. They are either read from the memory or computed from the *pc* register. Therefore, we want to get rid as soon as possible of values which are not instruction addresses.

However, this widening operator quickly leads to a lot of values approximated to $\top$. Usually, this is not an issue as we are not interested by most of computed data computations except when this value is the address handled by a store instruction: in this case, a $\top$ address would touch the whole memory. This means that all information collected by the analysis about the memory is scratched and lost. In turn, this would negatively impact the remaining

of the analysis. This is why the k-set analysis is useless alone: it is combined with another more precise value analysis (like CLP) such that, when an important value is required (address to load from, address to store to) and approximated by $\top$ in k-set, the matching CLP value is used instead and this usually leads to a much more precise analysed value.

Applying this method to the example of Fig. 1, the instruction at 0x2e260 may produce, for register r_3, the $\top$ value for k-sets and $(0, 1, 255)$ for CLP. If the condition `ls` of the comparison at 0x2e264 holds, r_3 becomes $\{0, 1, 2, 3\}$ for k-sets thanks to the CLP value $(0, 1, 3)$. From this, the address accessed at instruction 0x2e268 is $0x2e270 + \{0, 1, 2, 3\} \times 4$ and results in $\{0x2e270, 0x2e274, 0x2e278, 0x2e26c\}$. These are the exact set of addresses stored in the table used to translate the `switch`-case obtained by loading the words at the addresses provided by the k-set value.

3 Dynamic Branch and Real-Time Application

The approach presented in the previous section is effective but expensive to apply to a complete program. Therefore, we expose here a slicing method to speed up these analyses.

3.1 Analysis of the Whole Program

The functionality of real-time systems are often divided into tasks. The execution of the tasks are scheduled statically in the event loop or dynamically with the help of a real-time operating system. A task can be stand-alone, i.e. performing its functionality without depending on the outcome of the other tasks. On the other hand, a task might require the results of the others, through task communications [12]. The communication between the tasks relies on mechanisms such as globally shared variables and pointers, where a task writes to a variable and it is read by other tasks.

When analysing solely a task which reads from a globally shared variable, there will be no assumption made to the value of such variable, i.e. the writes to the variable are outside of the analysed task. Also, as stated in [7], to have a safe analysis (where all the possibilities are considered) of the function pointers, it is required to perform the analysis on the program as a whole. Indeed, a function pointer called in one task may be used by another task.

3.2 LightSlicing – a Smart and Effective Slicing Approach

To have safe results, tasks communicating with each other shall be analysed together. It is obvious to see that the complexity of the analysis grows as the number of tasks grows. Even though the amount of instructions to analyse grows, it can also be seen that some codes/instructions do not have influence over the results of the analysis. In this case, the technique *program slicing* [14] can be applied to remove the uninteresting codes. For example, in Figure 1, the lines 1 to 3 (as well as lines 8 to 13) do not affect the outcome of the branching, and they can be sliced away.

The problem now is that slicing a program is a costly operation in analysis time requiring data flow analyses and several graph constructions and are even more time-consuming operations applied to machine language. To be effective, the slicing time must not exceeds the gain in analysis time of the *dynamic* control instructions. It already exists an approach to perform fast slicing on machine code [10]. However, while the slicing is performed, the working elements (registers and memory locations of interest) continue to grow even if their content is no more relevant, which leads to keep unnecessary parts of the program. It is not efficient enough to significantly reduce the program CFG size.

Algorithm 2 LightSlicing algorithm.

```
1: K ← I
2: wl ← {(v, USE(i)) / i ∈ I ∧ i ∈ v ∧ v ∈ V}   ▷ consider BB containing instructions of I
3: while wl ≠ ∅ do
4:     (v, we) ← pop(wl)
5:     WE(v) ← we
6:     for all i ∈ reverse(v) do                  ▷ examine instructions of BB in reverse order
7:         if DEF(i) ∩ we ≠ ∅ then                ▷ if the instruction define a useful register
8:             K ← K ∪ {i}                        ▷ the instruction is kept
9:             we ← we \ DEF(i) ∪ USE(i)          ▷ its used registers are now interesting
10:        end if                                 ▷ yet its defined registers are no more interesting
11:        if i = FIRST_INST(v) then              ▷ when reaching the first instruction
12:            wl ← wl ∪ {(w, we ∪ WE(w)) | w ∈ pred(v) ∧ we \ WE(w) ≠ ∅}
13:        end if   ▷ the predecessors are added to wl if the fixed-point is not reached
14:    end for
15: end while
```

Hence, we introduce our slicing approach on binary code: the *LightSlicing*. It adapts the conventional *DEF and USE* approach used to build DU- or UD-chains described in [8]. The *DEF* and the *USE* give up a set of elements (i.e. registers and/or memory addresses) that an instruction writes a value to and reads a value from, respectively. Conventional program slicing based on this will start with a set of elements of interest, which we call *working elements, or we*. During the process, the instructions *inst* that write to any of these elements, donated by using the *DEF(inst)*, will be kept. Then, the redefined element(s) are removed from the working elements; while the required elements, i.e. the elements to read which are obtained by using the *USE(inst)*, are added to the working element. In general, identifying the registers in the *DEF* and *USE* is straight-forward. In contrast, obtaining the memory addresses can be complicated. For example, a memory address to read (or write) may be stored in a register which value is decided at run-time. Therefore, the help of *address analysis* is required. For a large program, to have a coherent result of the address analysis, the whole program must be taken into account, which may lead to an expensive computation time. To achieve better performance, LightSlicing does not require the address analysis: the whole memory is considered as a single register: we loose in precision but hope to gain a lot in speed. LightSlicing is applied just before each iteration of the dynamic branching resolution to avoid unnecessary computations and hence the speed-up. The details of LightSlicing is shown in Algorithm 2: the result is K, the set of instructions to keep while the initial set of interesting instructions is I.

4 Experiments and Findings

We carried out the experiments over two sets of benchmarks: the Mälardalen benchmarks [9], and the realistic industrial Engine Management System from Continental Corporation (which consists of 7 tasks, 184 KLOC). The experiments were carried out on Intel i7-4810MQ 2.8 GHz with 32 GB of RAMs. Because we are mainly interested in the detection of the unknown target addresses, due to the `switch`-cases and calls on function pointers, we only experimented with *cover*, *duff*, and *lcdnum* from the Mälardalen benchmarks. The results of the analysis times and speed-ups (due to applying LightSlicing) are shown in the Figure 2. Since the examples from the Malardalen benchmarks are much smaller than the industrial

case, we multiply the analysis time with 1000 to make them visible in Figure 2. Because we are interested in the impacts due to the application of the slicing, we performed the analyses for three different cases: (a) the analysis without program slicing, (b) applying program slicing with address analysis, and (c) the analysis with *LightSlicing*. We are able to resolve all the dynamic branches of the Mälardalen benchmarks, but 92% for the industrial example. It is mainly due to that the slicing of the industrial application makes irreducible loops that are not completely handled by OTAWA framework, hence we can not perform the analysis on the whole program but on its individual tasks.

Figure 2 shows the distributions of time taken by each analysis. For the analysis without slicing, we use the real execution time (in micro-seconds) for the vertical axis. To compare the performance for the analyses that take advantage of the slicing, we use the speed-up as the vertical axis. The speed-up is calculated by Equations 4. When the value of the speed-up is less than one, this means that the analysis runs slower than the non-slicing approach: we use red lines in the figure to represent this boundary.

It shows that the analysing times decrease drastically, with the average of 7.30 times, and the maximum of 33.37 times of speed-up, when applying LightSlicing. From (a) we can see that the CLP analysis takes the majority of the analysis times because it is more complex than the k-set analysis. Since the address analysis used in (b) is implemented within the CLP analysis, which leads the CLP analysis to take more proportion in the analysis. Because the size of the CFGs are reduced, the time spent on k-set analysis is also reduced. We can also observe that the slicing does not impact overall performance heavily. In (c), both of the CLP and k-set analyses are performed upon the reduced CFGs, as the result from applying LightSlicing and hence the reduction of the analysis time. LightSlicing takes more proportion in the analysis because it works on the full CFGs, however it reduces the analysis by large amounts for all cases.

It also shows that having slicing with address analysis may have negative impact on the performance for smaller examples, whose speed-ups are less than one (in the grey-out area). This is because the overhead from the address analysis can not be compensated by the time saved due to slicing. We avoid the address analysis in LightSlicing and obtained the improvements up to 33.37 times faster. LightSlicing works particularly well on larger codes which justifies its use in realistic and real-time applications.

$$Speedup = \frac{T_{non\text{-}sliced}}{T_{sliced}} \tag{4}$$

5 Related Works

Building CFGs from binaries is a recurrent issue for static analysis of binaries, for making smart debugger or for reverse-engineering programs [15]. Theiling, in [13], proposes a multi-instruction set generic framework to extract CFGs from binaries. He identifies the issues in the determination of *dynamic* branches but no solution is provided.

Bardin et al. in [4] use variable-precision k-sets to compute the targets of *dynamic* branch instructions. The k determination is variable for each handled value and adjusted according to the need of precision, focused in this case, on the set of possible targets of a branch. The experimentation on an industrial application (21 kloc of C) exhibits relatively long computation times (in tens of minutes). Moreover, the authors do not address the problem of memory loss due to imprecise k-set values.

In [3, 1, 2], Balakrishnan and Reps present a complete method to perform data flow analysis, resolve *dynamic* branches and extend the CFG in an incremental way. The approach

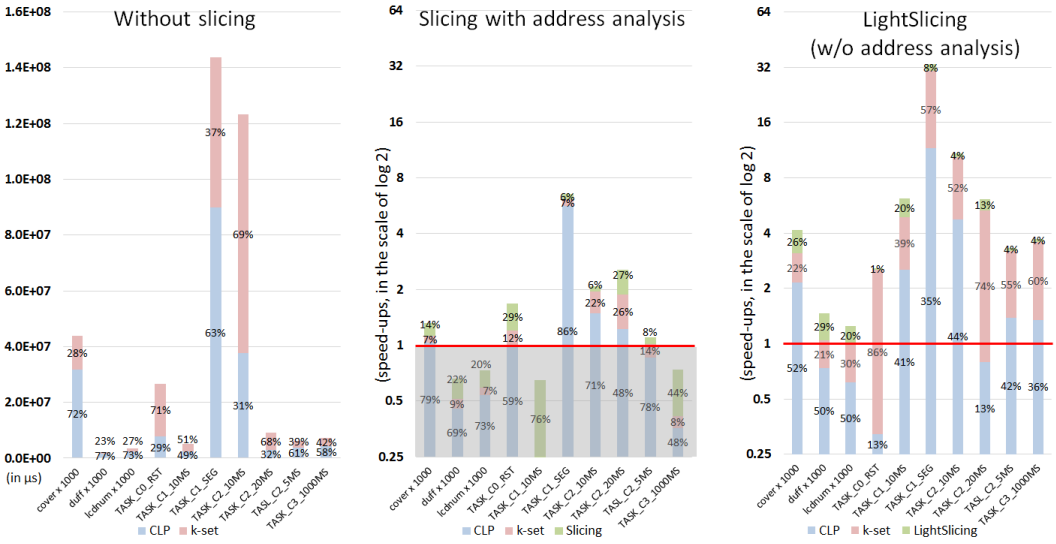


Figure 2 Execution comparisons of the dynamic branching analyses.

is quite integrated and therefore relatively costly in analysis time. Yet, they use, as abstraction of values, a form close to CLP and hence suffers from the over-sampling problem.

In a recent article [7], Holsti et al. experiments and compares several common value analyses (including CLP) to resolve the target of *dynamic* control instructions. They identify several shortcomings in these classic value analyses (particularly the over-sampling problem of CLP) and the requirement to analyse the whole application for function pointers resolution inside tasks of a real-time system. None of the experimented analyses overtakes the others but their limitations are highlighted.

6 Conclusion

In this paper, we have presented an approach to resolve *dynamic* control instructions. The approach is based on a usual value analysis (CLP in this case but this could be another value analysis) used to help the k-set analysis. The k-set analysis enables us to precisely preserve the possible target addresses of branch instructions. In the case of function pointers, it is shown in [7] that a whole analysis of the application is needed. As this analysis may be time-consuming, we propose a fast slicing method which works on the machine codes and speed up the subsequent value analyses.

The experiments conducted on a subset of Mälardalen benchmarks and on a real industrial application shows good but not perfect results. The main cause of unresolved *dynamic* branches is the precision of the auxiliary value analysis (CLP): in future works, we plan either to improve our CLP analysis, or to replace it with a better value analysis, or to combine together several value analyses providing different aspects of the program values.

Then, although we have achieved very good analysis time, particularly on the real industrial application, it remains some room for improvement: at each step of the analysis, CLP and k-set analyses are wholly re-computed while only a small part of the CFG is changed. A good effect of Abstract Interpretation based analyses is that the detailed behaviors of each instruction/BB are simplified and abstract states are used to present the effects constituted by each part of the CFG. The changes in the abstract states propagate throughout the CFG

and new paths of propagation could be formed according to the evolution of the abstract states. If the propagation of new paths does not contribute a lot of time to the overall analysis, we expect substantial speed-ups from the incremental calculation of the CLP and the k-set analyses for the calculation of *dynamic* branches targets.

References

1. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. *Compiler Construction: 14th International Conference, CC 2005*, chapter CodeSurfer/x86 – A Platform for Analyzing x86 Executables, pages 250–254. Springer Berlin, 2005.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 2732–2733. Springer Berlin, 2004.
3. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *Verified Software: Theories, Tools, Experiments*, chapter WYSINWYX: What You See Is Not What You eXecute, pages 202–213. Springer Berlin Heidelberg, 2008.
4. S. Bardin, P. Herrmann, and F. Védrine. *Refinement-based CFG reconstruction from unstructured programs*, pages 54–69. Springer, 2011.
5. Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture value analysis for machine code. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, pages 42–52, 2013. doi:10.4230/OASIcs.WCET.2013.42.
6. 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*, pages 238–252. ACM Press, 1977.
7. Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing switch-case code with abstract execution. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 85–94, 2015. doi:10.4230/OASIcs.WCET.2015.85.
8. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
9. WCET Project Mälardalen University. Benchmarks. URL: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
10. C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET Flow Analysis by Program Slicing. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED*, pages 103–112. ACM, 2006.
11. R. Sen and Y. N. Srikant. Executable Analysis with Circular Linear Progressions. Technical Report IISc-CSA-TR-2007-3, Computer Science and Automation Indian Institute of Science, February 2007.
12. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.
13. H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. of 7th Conference on Real-Time Computing System and Applications*, 2000.
14. M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.
15. W. Yin, L. Jiang, Q. Yin, L. Zhou, and J. Li. A control flow graph reconstruction method from binaries based on XML. In *Computer Science-Technology and Applications, 2009. IFCSTA'09. International Forum on*, volume 2, pages 226–229, Dec 2009.

Measurement-Based Timing Analysis of the AURIX Caches*

Leonidas Kosmidis[1], Davide Compagnin[2], David Morales[3], Enrico Mezzetti[4], Eduardo Quinones[5], Jaume Abella[6], Tullio Vardanega[7], and Francisco J. Cazorla[8]

1 **Universitat Politècnica de Catalunya, Barcelona, Spain; and Barcelona Supercomputing Center, Spain**
2 **University of Padova, Padova, Italy**
3 **Barcelona Supercomputing Center, Barcelona, Spain**
4 **Barcelona Supercomputing Center, Barcelona, Spain**
5 **Barcelona Supercomputing Center, Barcelona, Spain**
6 **Barcelona Supercomputing Center, Barcelona, Spain**
7 **University of Padova, Padova, Italy**
8 **Barcelona Supercomputing Center, Barcelona, Spain; and IIIA-CSIC, Barcelona, Spain**

Abstract

Cache memories are one of the hardware resources with higher potential to reduce worst-case execution time (WCET) costs for software programs with tight real-time constraints. Yet, the complexity of cache analysis has caused a large fraction of real-time systems industry to avoid using them, especially in the automotive sector. For measurement-based timing analysis (MBTA) – the dominant technique in domains such as automotive – cache challenges the definition of test scenarios stressful enough to produce (cache) layouts that causing high contention. In this paper, we present our experience in enabling the use of caches for a real automotive application running on an AURIX multiprocessor, using software randomization and measurement-based probabilistic timing analysis (MBPTA). Our results show that software randomization successfully exposes – in the experiments performed for timing analysis – cache related variability, in a manner that can be effectively captured by MBPTA.

1998 ACM Subject Classification D.4.7 Real-time Systems and Embedded Systems

Keywords and phrases WCET, caches, AURIX, Automotive

Digital Object Identifier 10.4230/OASIcs.WCET.2016.9

1 Introduction

Despite the complexity added by caches to timing analysis [11], their potential benefits in the reduction of WCET estimates have motivated their analysis for decades [27]. In terms of

* The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (`http://www.proxima-project.eu`), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation (grant TIN2015-65316-P) and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship RYC-2013-14717. This work was conducted as part of a collaboration with the CONCERTO project (ARTEMIS-JU grant nr. 333053), which provided the automotive application and the build automation. Authors thank Benjamin Lesage for his support in execution time collection infrastructure on the AURIX board.

performance, it is well known that the program memory layout, i.e. the addresses in which the program's code and data are located, determines the program's *cache layout*, i.e. the cache sets to which the program's data and code are mapped. For a given program, the execution time variability under different cache layouts can be significant, ultimately affecting processor performance [22][25][23]. This occurs because both even small variations in the order in which the object files are linked together [25] and in other elements of the memory layout (e.g. environmental variables) may significantly affect execution time. This variability in execution time complicates timing analysis significantly. In confirmation of the difficulties that caches bring into timing analysis, the state of the art chips for the automotive market typically include large scratchpads (32KB for code and 120KB for data in our target platform) and relatively smaller caches (16KB for code and 8KB for data in our target platform), possibly disabled by default. However, in the general case, transparently-managed caches offer a more flexible and efficient hardware acceleration mechanism than user-controlled scratchpad. The computational power required to support the user demands for increasingly complex functionality suggests that in the long term, scratchpads and caches will coexist in embedded real-time systems. It therefore stands to reason that cache-related variability needs be characterized and analysed.

Measurement-based timing analysis (MBTA) methods are widely used in application domains such as automotive [27][1]. MBTA application comprises the *analysis phase* in which verification of the timing behaviour is performed and the *operation phase* in which the system is deployed in the actual operational environment. MBTA aims at estimating a WCET estimate that holds during operation with execution-time measurements taken at analysis time. In the context of MBTA, the challenge of using caches lies on providing evidence that, in the measurements runs, the cache layouts that lead to high execution times are properly factored in when computing the WCET estimates. However, in general, it is hard for the user to design experiments in which bad (worst) cache layouts are enforced. This reduces the confidence on the WCET estimates obtained with MBTA required for timing verification according to the domain-specific safety standards, resulting in the cache not being fully embraced by the real-time industry.

Performing an exhaustive exploration of the space of potential cache layouts is not only practically difficult (assuming full control of hardware and software states) but also computationally infeasible in the general case. Cache randomization techniques [14][18] may help in this respect by ensuring that a new random cache layout is exercised in every program run. As a result, the search space of cache layouts is transparently (and randomly) explored by performing additional runs of the program at analysis time. This, in turn, allows deriving a probabilistic characterization of the impact of changes in the cache layout on the execution time of a program. Furthermore, such probabilistic characterization facilitates the application of Measurement-based Probabilistic Timing Analysis (MBPTA) [9, 1] in that it probabilistically guarantees that those cache layouts leading to higher execution times have been captured at analysis time.

Two main flavours of cache randomization exist: hardware randomization employs time-randomized caches [13], featuring random placement and replacement policies; software randomization [18] instead randomizes the memory layout of the program's data and code

[1] While the scientific literature extends on trade-offs between static and measurement-based approaches [3], industrial practitioners take a pragmatic view to when to use either, which considers cost-effectiveness in achieving required evidence. This paper contributes how to increase confidence on measurement-based approaches – massively used in automotive – in the presence of caches.

across runs. In this paper we present our experience with applying MBPTA and software-based cache randomization to probabilistically characterize and analyse the impact of code and data layout of an Automotive Cruise Control System (ACCS) running on an AURIX TC277 processor board [26]. In this work is we do not provide an analysis method for the TC277, which has been designed to be with determinism in mind. Nor we make a comparison between MBPTA and static timing analysis techniques, which has been already done [3]. Instead, in this paper we demonstrate that the impact of cache layout on execution time variability is relevant even on deterministic architectures, showing how this variability can be effectively analysed and characterized with randomization and MBPTA.

2 Background and Problem Statement

In the presence of caches, the memory layout of a program impacts the pattern of hits and misses – and hence the timing behaviour – of individual program runs [7]. Even small changes in their memory layout may cause significant jitter in the observed timing behaviour [19]. The memory layout is subject to small – many times transparent-to-the-user – changes (e.g., compiler flags or even small code modifications). As a result, no deterministic timing analysis approach can provide trustworthy timing guarantees unless either the memory layout is guaranteed not to change or the worst-case layout is determined (and observed, in MBTA approaches). However, freezing the memory layout is typically not possible before the very end of the development process and the definition of a global (best-) worst-case memory layout for an application comprising several tasks is a generally intractable problem. Needless to say, changes to the memory layout may always occur even after deployment, as a result of code patches and other modifications.

MBPTA [9, 1] offers a solution to probabilistically characterise the impact of memory layout on the timing behaviour of a program while still exhibiting the appealing cost-benefit ratio of deterministic measurement-based methods [27]. Probabilistic WCET (pWCET) estimates computed with MBPTA differ from standard WCET figures in that they do not consist on a single value but corresponds to a probability distribution of high execution times. The pWCET distribution conservatively models the residual risk for *one instance* of the target program to exceed a given execution time bound. Users are then interested in those execution time bounds whose exceedance probability is considered acceptable in relation to the integrity level of the functionality being analysed, which in turns depends on the corresponding safety standard.

MBPTA builds on extreme value theory [17] (EVT) to model the pWCET distribution. EVT requires that the observed execution times of the program under analysis must be modellable with *independent and identically distributed* (i.i.d.) random variables. This requirement has been shown to be achievable with the adoption of MBPTA-compliant hardware [15]. In the absence of time-randomised hardware, by exploiting specific software techniques. In particular, software randomisation (SWRand) approaches have been shown to enable the analysis of deterministic caches with MBPTA [14][18]. Different software randomization variants are presented in Section 5.

Interestingly MBPTA and EVT are not the same thing [8]: while EVT can be, in principle, applied to time-deterministic architectures, it would still requires the user to provide evidence that the measurements at analysis time capture the potential variability at operation time. This same requirement, instead, is almost met by construction in MBPTA-compliant platforms where the sources-of-variability are operated almost transparently to the user (or with low user intervention) in a way that guarantees that the *jitter they cause at analysis matches or upperbounds that which may occur during operation* [15].

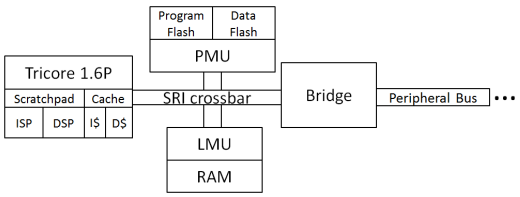


Figure 1 Block Diagram of one core and the LMU/PMU in the AURIXTC277.

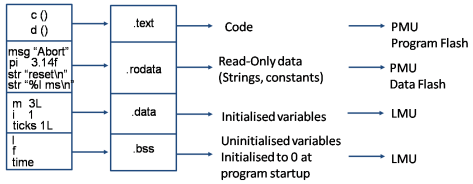


Figure 2 Memory mapping configuration used for the application.

- *Deterministic upperbounding.* Some resources are forced to work on their worst latency so that the analysis time measurements capture the worst timing behaviour that those resources may have during operation [15]. This can be applied to resources with limited impact in the WCET (e.g., some floating point operations with input-dependent latencies).
- *Probabilistic upperbounding.* The previous approach is unaffordable for resources in which jitter is high, since assuming every access to incur the longest latency would cause a significant performance degradation. Instead resource timing randomization and probabilistic reasoning is used. For instance, the reliability of the upperbounds derived by EVT over deterministic caches lies on the user ability to design experiments that exercise conflictive cache layouts causing high execution times. With hardware randomization [13] or software randomization [14], the space of cache layouts is randomly sampled in every new run. Hence, the user only cares about the number of runs to perform instead of having to enforce specific cache layouts.

It is worth noting that MBPTA is exposed to the risk of not capturing the representativeness of events with significant impact on execution time and low probability [2][24]. As discussed in [2][20][21], it is possible to reduce the risk that any such timing event has not being captured in the execution time measurements used by MBPTA. Further exploring this topic is part of our future work.

3 Target AURIX Platform

For this work we use an AURIX TC277 board [26]. Although it is a three-core architecture (TriCore), in this study we focus on a single-core configuration in order to isolate the cache effects from contention-related jitter [29]. That is the application described in Section 4 is executed on one core, while the other two cores remain idle. As part of our future work we aim at combining the solutions to simultaneously handle cache jitter and contention jitter.

The AURIX platform has been designed to have as few jittery resources as possible, following current industrial practice for timing analysis based on determinism. In particular, all the three cores present in the platform are equipped with scratchpad memories to provide deterministic memory accesses. As a platform designed for reliability, the AURIX TC277 is a heterogeneous platform comprising differently implemented cores, which share a common ISA. The first core, which is optimised for low-power execution, has a simpler microarchitecture than the other two high-performance cores, which are equipped with high-performance features such as caches, which create jitter. Our goal is to show how caches in an automotive system like the one presented in this paper can be handled with randomisation and MBPTA. Cores are connected to the 'memory system' through the Shared Resource Interconnect (SRI), see Figure 1. The memory system comprises a SRAM device (LMU - Local Memory Unit) and FLASH device (PMU - Program Memory Unit). The 32 KB SRAM shared memory can

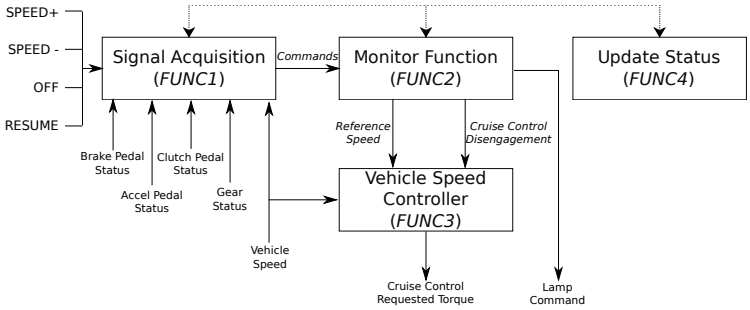


Figure 3 Case Study application overview.

be defined through the LMU as cacheable or uncacheable. Similarly, the PMU flash memory can be also configured as cacheable or not, and is divided in separate units for code and data. The application/RTOS defines statically in a linker script which application software element (function, data etc) is mapped in which hardware resource (LMU, PMU, scratchpad, CPU number etc) and whether accesses to it will be cached or not. Each cacheable memory access can result in a hit or miss in cache, which creates a source of variability. This is handled in the target AURIX platform by MBPTA and static software randomisation (SSR, see Section 5 for further details).

In the integration of SSR in our target AURIX platform we note that AURIX implements a physical memory management unit in which the memory address used in read and write operations determines the device location in which data reside and their cacheability. Concretely, AURIX defines 15 memory segments defined during the linker process, specifying that objects must be located in the LMU, PMU, scratchpads or caches. In order for the timing effects of SSR, as described in Section 5, to be effective, memory objects need to be located in cacheable memory segments: concretely in segments 8 and 9, which allow cached access to PMU and LMU respectively, as shown in Figure 2. For our experiments, the application is executed on one of the high-performance cores, which features caches.

4 Automotive Cruise Control System (ACCS) Case Study

The real-world application analysed in this work implements an Automotive Cruise Control System (ACCS), automatically generated from a Simulink model. The functional code generated by Simulink has been merged to the architectural code obtained by model transformations applied to the CONCERTO representation of the original application model[2]. The application has been adapted to run on top of a customized version of ERIKA[3], a OSEK/VDX compliant Real-time Operating System, which has been modified to exhibit time-composable timing behaviour [4]. The application includes stubbing for IOs to facilitate the analysis and keeps iterating over a discrete and finite set of inputs. While this may limit the realism in the observation of the application behaviour, the resulting execution scenario is acceptable for the intent of our work.

As depicted in Figure 3, the application consists of four main tasks:

- *Signal Acquisition*: it provides signal acknowledgement to the system. It periodically

[2] CONCERTO, ARTEMIS JU, `http://www.concerto-project.org/`.
[3] Erika Enterprise RTOS, `http://erika.tuxfamily.org/drupal/`.

reads and converts the inputs (i.e., the pedals, the cruise user controls and the vehicle speed) and provides them to the Monitor Function task;

- *Monitor Function*: it implements a finite-state machine for the ACCS. It computes the state transition when activated by the Signal Acquisition task;
- *Vehicle Speed Controller*: it performs several interpolations to compute the requested torque according to the system state and the vehicle speed. Its execution is triggered by the Monitor Function task;
- *Update Status*: it updates the inputs according to the current execution time.

The application has been deployed on the AURIX target according to a specific configuration where the global data and the task's private stack are allocated in the cached SRAM. Instructions are stored in the PFlash memory and accessed from the cacheable segment. Input and outputs were stubbed to dispense with the use of actual sensors and actuators.

5 Software Randomization

The main characteristic of cache software-randomization techniques is that they enable the use of MBPTA on COTS, i.e. non-hardware randomized, caches. Software randomization techniques help assessing that the jitter generated by the cache in analysis phase exposes the jitter that the cache can generate at operation phase. To the best of our knowledge, software randomization is the only technique that enables the analysis of COTS caches with MBPTA. There are two main variants of software randomization: dynamic and static.

Dynamic Software Randomisation (DSR) [14][10] performs the randomisation at runtime during the initialisation phase of the program, so that the location of objects in memory is randomised across different executions of the program. DSR combines a compiler pass that modifies appropriately the intermediate representation of the application's code and a run-time system, based on self-modifying code, that is in charge of performing the relocation of objects in memory. The memory objects whose location is randomised are code, stack frames and global data. However, it has been shown [18] that DSR generated code makes intensive use of pointers and dynamic objects which complicates its use in automotive, where the ISO26262 [12] standard requires a limited use of pointers and no use of dynamic objects (among others). Moreover, most automotive processors and microcontrollers, including AURIX, impose practical limitations which prevent the use of self-modifying code, such as fetching code and read-only data directly from read-only flash devices, featuring small dynamic memories (LMU) and strong memory protection units.

With *Static Software Randomisation* (SSR) [18] instead, the program code, stack and global data are allocated to random memory positions across images, achieving the same effect as DSR in an entirely static manner. SSR builds on the fact that memory objects in the executable defines their placement in main memory, and therefore the cache layout. SSR carries out all relocation operations statically at compile time. SSR generates a number of different binaries of the same program each with different random allocation of memory objects in the executable. By randomly selecting an executable from the pool of the generated ones, the timing properties required for the pWCET estimates are preserved. Similarly to DSR, SSR randomises code (functions), stack frames and global data. Interestingly SSR is certification aware since it does not use pointers [18]. For these reasons, and because of the practical limitations of applying DSR on AURIX, this paper focus on SSR.

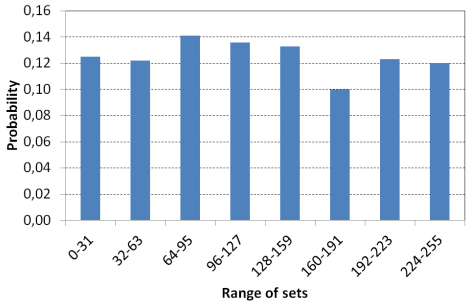


Figure 4 Cache Set distribution of the first instruction of FUNC1 in set groups.

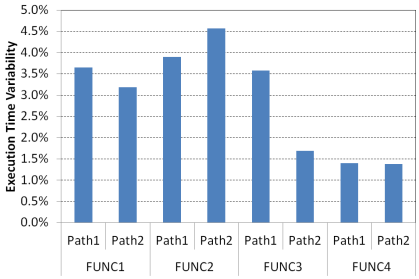


Figure 5 Observed Jitter across runs of the different FUNC for each path.

6 Experimental Results

In the AURIX platform, the execution time of ACCS may be influenced by:

(i) the initial processor state when the task starts its execution,
(ii) the interference from the underlying RTOS,
(iii) the time randomisation injected by some processor resources and
(iv) the task input data.

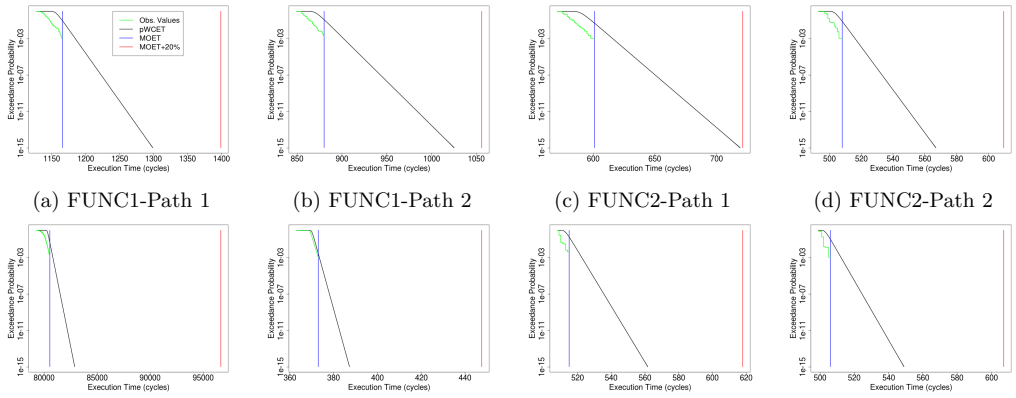
All the above factors are taken into account in our experiments by guaranteeing that the execution conditions enforced at analysis time over-approximate those at deployment. We cover issue (i) by flushing the cache state before each experiment: we start collecting measurements only after the cache warms up and execution times stabilize, which occurs after very few iterations. With respect to factor (ii), we guarantee the OS-induced jitter to be probabilistically analysable [4] by using a time-composable version of the ERIKA [29], as noted in Section 3. The jitter introduced by SSR is also taken into account by MBPTA [9, 1], thus covering issue (iii).

In terms of input-related variability (iv), the MBPTA variant we use [9, 1] can only reason about the paths triggered by the test runs performed at analysis. Solutions to deal with path coverage in MBPTA exist [16][28] but they are not required in this specific context as we rely on a controlled set of input vectors, leading to the different paths. In fact, in our evaluation we focused on two specific paths of the ACCS application, corresponding to the worst-case and best-case observed paths for the used input vectors. We refer to those paths as `Path1` and `Path2` respectively and reason on pWCET estimates obtained separately for each path.

6.1 Randomisation Assessment

We validate that SSR uniformly randomises the cache layout by studying the distribution of the first instruction of the functions under analysis, in the instruction cache sets. Since all the functions under analysis have the same behaviour, in Figure 4 we provide this information for only one of them, FUNC1. Given the initial address of the function, recorded across 1,000 runs, we determine the probability that the first instruction of the function is mapped to any group of 32 sets (0–31, 32–63, ..., 224–255). As it can be observed SSR achieves a fairly balanced distribution across runs, with similar results obtained for other group sizes. Note that for a higher number of runs, the probability of each group converges to 1/8=0.125, due to the uniform random nature of SSR. This evidences that SSR randomises effectively the cache memory layout.

Since the basic principle of SSR is that the memory layout has a direct impact on the execution time, we also assess the effect of cache layout randomisation on the execution time.



(a) FUNC1-Path 1 (b) FUNC1-Path 2 (c) FUNC2-Path 1 (d) FUNC2-Path 2

(e) FUNC3-Path 1 (f) FUNC3-Path 2 (g) FUNC4-Path 1 (h) FUNC4-Path 2

Figure 6 pWCET curves derived with MBPTA for the paths of FUNC1-FUNC4.

Table 1 Independence and identical distribution tests results for FUNC1-4.

	Path1		Path2	
Test	i.d.	ind.	i.d.	ind.
FUNC1	0.81	0.89	0.96	0.13
FUNC2	0.59	0.55	0.85	0.10
FUNC3	0.28	0.92	0.89	0.51
FUNC4	0.85	0.63	0.24	0.51

Table 2 WCET reduction between MBTA and MBPTA for cut-off probability 10^{-12}.

	Cut-off prob. 10^{-12}	
	Path1	Path2
FUNC1	9%	9%
FUNC2	5%	11%
FUNC3	15%	14%
FUNC4	11%	11%

In Figure 5 we observe the variability induced by SSR on the execution time of each observed path of the functions under analysis. This variability (jitter) across executions of different binaries is between 1.5–4.5%. Note that the observed variability is introduced only from the instruction and data cache. Recall that due to the deterministic nature of AURIX executions with the same binary and input do not have any variability. Moreover, since the execution times are collected per path and the same input is used, the variability does not come from the traversal of different paths. Hence, SSR can be used in order to provide insights to the industrial user about the potential impact of memory layout in its final integrated system. Further, from the above results we can conclude that SSR exposes cache jitter impact into execution times, whose probabilistic properties are examined in the next Section.

6.2 Independence and Identical Distribution Tests

We execute the case study 1,000 times to collect execution times, each with a different binary generated by SSR by using different random seeds. We flush caches and reload the executable across executions to have the same clean environment for each execution. This setup complies with MBPTA requirements [15]. The application of EVT – part of MBPTA – requires the execution times (data) provided as input to be statistically i.i.d. We test independence with the Ljung-Box test [6]. For identical distribution we use the two-sample Kolmogorov-Smirnov test [5]. In both cases we use a 5% significance level (a typical value for this type of tests). These means that i.i.d. is rejected only if the value for any of the tests is lower than 0.05. As shown in Table 1 all tests are passed in all setups, hence enabling the application of EVT.

6.3 pWCET Estimates

In Figure 6 we show pWCET estimates for every function and path. In all cases we successfully derived a pWCET curve, whose slope changes from case to case. The Complementary Cumulative Distribution Function (CCDF) of the observed execution times is also reported (shorter scattered green line). The fact that the CCDF is always below the pWCET curve confirms that in all cases our observations are tightly upperbounded by the pWCET.

In addition to the pWCET curves, we also show the maximum observed execution time, or MOET, for each FUNC and path (vertical line on the left), and a WCET estimate computed with MBTA using an engineering margin of 20% (vertical line on the right), commonly used in the real-time industry (e.g. avionics), to account for systems unknowns that could not be fully exercised during analysis such as the memory and cache layouts[4]. Interestingly, in all the cases the pWCET obtained for exceedance probability 10^{-15}, is below the estimates computed with current practice. In particular, considering a cut-off probability of 10^{-12} which has been shown to be an appropriate exceedance probability for hard-real time systems [18], MBPTA provides between 6–17% tighter results than MBTA (see Table 2). Despite the WCET reduction yielded by MBPTA, its most important benefit is that it provides scientific reasoning about the pWCET upper-bound instead of an engineering margin.

6.4 Summary

Cache jitter can be arguably hard to observe with standard MBTA approach. This is so because cache layout is (mostly) implicitly handled by the RTOS and the end user lacks evidence that despite it makes high number of runs, the space of potentially cache layouts that can arise at operation emerge. With software randomization instead, every new run a random cache layout is generated, which makes that as more runs are performed the space of potential cache layout are naturally covered. This not only provides a coverage criteria but also simplifies the analysis of program high execution times with measurement based probabilistic timing analysis. In the ACCS study randomization (i) exposes that variability is reduced; and (ii) enables MBPTA. This results in a proper handling of cache jitter in WCET estimates and further provides evidence for certification.

7 Conclusions

We presented our experience in using a software-based cache randomization techniques – that randomizes across image runs the position in memory where program's data and code are located, and MBPTA for an automotive cruise-control systems on the AURIX TC277 board. We have shown that in such deterministic architecture like the AURIX, caches can cause execution time variability. We further show that cache jitter can be handled with randomization – that makes it naturally emerge in the runs performed at analysis time – and probabilistic timing analysis that models the probability of high execution times.

References

1 Jaume Abella. Improving mbpta with the coefficient of variation. Technical Report UPC-DAC-RR-CAP-2015-11, UPC, July 2015.

[4] The unknowns also cover other factors such as the jitter due to different paths, that with MBPTA can be factored in with solutions like [16][28].

2 J. Abella et al. Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.

3 J. Abella et al. On the comparison of deterministic and probabilistic WCET estimation techniques. In *ECRTS*, 2014.

4 A. Baldovin et al. A time-composable operating system. In *WCET Analysis Workshop*, 2012.

5 S. Boslaugh and P.A. Watters. *Statistics in a nutshell*. O'Reilly Media, Inc., 2008.

6 G.E.P. Box and D.A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *J. of the American Statistical Assoc.*, 1970.

7 L.P. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *SIGMETRICS Perform. Eval.*, 3(27):28–42, 1999.

8 F.J. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET Workshop*, 2013.

9 L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

10 C. Curtsinger and E.D. Berger. STABILIZER: statistically sound performance evaluation. In *ASPLOS*, 2013.

11 E. Mezzetti et al. Cache Optimisations for LEON Analyses (COLA). Technical report, ESA/ESTEC, 2011.

12 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

13 L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

14 L. Kosmidis et al. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.

15 L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *DSD*, 2014.

16 L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.

17 S. Kotz et al. *Extreme value distributions: theory and applications*. World Scientific, 2000.

18 L. Kosmidis et al. Containing timing-related certification cost in automotive systems deploying complex hardware. *Best Paper Award, DAC*, 2014.

19 E. Mezzetti and T. Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*, 2013.

20 E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.

21 S. Milutinovic et al. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *ISORC*, 2016.

22 N. C. Gloy et al. Procedure placement using temporal ordering information. In *MICRO*, 1997.

23 Eduardo Quinones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–138, 2009.

24 J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1), 2014.

25 T. Mytkowicz et al. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

26 `http://www.ehitex.de/application-kits/infineon/2531/aurix-application-kit-tc277-tft`. *AURIX Application Kit TC277 TFT*. hitex.

27 R. Wilhelm 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.

28 M. Ziccardi et al. EPC: extended path coverage for measurement-based probabilistic timing analysis. In *RTSS*, 2015.

29 M. Ziccardi et al. Software-enforced Interconnect Arbitration for COTS Multicores. In *WCET Analysis Workshop*, 2015.

Employing MPI Collectives for Timing Analysis on Embedded Multi-Cores

Martin Frieb[1], Alexander Stegmeier[2], Jörg Mische[3], and Theo Ungerer[4]

1 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
martin.frieb@informatik.uni-augsburg.de
2 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
alexander.stegmeier@informatik.uni-augsburg.de
3 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
mische@informatik.uni-augsburg.de
4 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
ungerer@informatik.uni-augsburg.de

Abstract

Static WCET analysis of parallel programs running on shared-memory multicores suffers from high pessimism. Instead, distributed memory platforms which communicate via messages may be one solution for manycore systems. Message Passing Interface (MPI) is a standard for communication on these platforms. We show how its concept of collective operations can be employed for timing analysis. The idea is that the worst-case execution time (WCET) of a parallel program may be estimated by adding the WCET estimates of sequential program parts to the WCET estimates of communication parts. Therefore, we first analyse the two MPI operations MPI_Allreduce and MPI_Sendrecv. Employing these results, we make a timing analysis of the conjugate gradient (CG) benchmark from the NAS parallel benchmark suite.

1998 ACM Subject Classification C.1.2 [Processor Architectures] Multiple-Instruction-Stream, Multiple-Data-Stream Processors (MIMD), C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems, D.1.3 Concurrent Programming, D.2.13 [Reusable Software] Reusable libraries, J.7 [Computers in Other Systems] Real time

Keywords and phrases Real Time, Network on Chip, WCET, Timing Analysis, MPI

Digital Object Identifier 10.4230/OASIcs.WCET.2016.10

1 Introduction

Future embedded real-time systems might realise high performance through high parallelism with many cores. For increasing core numbers, shared memory puts strong limitations on static WCET analysis: it always has to be assumed that all nodes access the memory before the own request is processed by the memory controller (cf. [12, 5]). Furthermore, modern multi- and manycore architectures employ networks on chip (NoCs) to connect cores.

Therefore, parallel platforms with distributed memory and message-based communication might be the way to go (cf. [8]). Message Passing Interface (MPI) [4] is the standard for message-based communication which is widely used in high-performance computing. It encapsulates communication not only in single requests, but also in collective operations,

where all participating cores work together. For example, one (master) core distributes data and coordinates computation which is done by several (slave) cores. This is achieved by executing the same code on all cores and making case distinctions at some points to differentiate between master and slave cores. Moreover, this characteristic is beneficial for timing analysis of a parallel program: worst-case response times (WCRTs) are low and all cores share the same state in the program. Communication steps work as implicit barriers: Cores synchronise and afterwards go on executing sequential code.

However, only few research already took place in the field of real-time message passing parallel programs. Metzlaff et al. [9] described that an overall WCET estimate of a parallel program may be determined by adding the WCET estimate of communication parts to the WCET estimates of sequential program parts. They presumed a predictable NoC, simple cores and distributed memory. Now we go one step further: instead of only determining the worst-case traversal times (WCTTs) for the communication, the WCETs of MPI collectives including the WCTTs are estimated. Then, these collectives' WCET estimates can be added to the WCET estimates of sequential program parts to get an overall WCET estimate.

Since MPI collective operations can be reused by *any* MPI program, the corresponding timing analysis may also be reused. Thus, the analysis effort for MPI collective operations is necessary only once for a given hardware platform and WCTT behaviour (schedule, see Section 2.3). Then, the only remaining work at new programs is to WCET analyse the sequential program parts.

The contributions of our paper are the following: First, we show that a generic timing analysis of MPI collectives is possible for a given hardware platform and generic schedule. We formulate equations with hardware- and schedule specific parameters which can be used to determine the WCET estimate of a MPI operation. Then, we utilise these equations to determine the WCET estimate of a benchmark, where we determined sequential WCET estimates and joined them with WCET estimates of MPI collective operations.

The rest of the paper is structured as follows: In the next Section, we present related work and backgrounds about (real-time) MPI and the setting for our analysis. Afterwards, we analyse MPI collective operations in Section 3 and utilise the results for the analysis of the conjugate gradient (CG) benchmark in Section 4. Finally, we conclude our paper in Section 5 and give an outlook to future work.

2 Related Work and Background

2.1 MPI

MPI is the de-facto standard for message passing [4]. It encapsulates all communication between cores or distributed systems. MPI programs are typically written in a way that all cores execute the same code.

Many MPI programs utilise collective operations to distribute a computation and gather results. These are operations which are executed by all cores of a group[1]. A simple example would be a barrier, but data exchange also often works with collective operations, e. g. MPI_Scatter or MPI_Gather. MPI_Scatter distributes data stored at one core to all participating cores. Afterwards, each core has an equally sized portion of data to work with. MPI_Gather is its counterpart – it collects data from several cores to compound it.

[1] At MPI, groups help to specify which cores communicate together. A group may have any size – only two cores, but also all cores.

The first step towards a timing-analysable MPI was MPI/RT, a standard for real-time MPI [6, 14], an extension of MPI 1.1 from 1998. On the one hand, its extensions are quite broad. On the other hand, MPI/RT is quite old, does not respect NoCs and is not commonly used. In our implementation we focused on being simple and time-predictable and on following the widespread standard MPI [4].

In the context of real-time multi- and manycore architectures, Sørensen et al. [16] already analysed simple MPI operations for the Argo NoC [7]. They implemented send and receive operations as well as barriers and broadcasting. In our paper, we focus on collective operations and their impact on timing analysis. Moreover, we develop a concept for the complete timing analysis of MPI programs. Furthermore, we apply it on a complete benchmark and compare two different variations of time division multiplexing (TDM).

2.2 PaterNoster NoC and our Implementation

For being able to estimate the WCET of an MPI operation, it is required that the underlying NoC ensures upper bounds for communication. In our paper, we use the PaterNoster NoC [10] which fullfills this requirement by providing *guaranteed service* (GS) with TDM [11], see details in Section 2.3.

The nodes in our NoC are arranged in a quadratic torus and connected via unidirectional X rings (horizontal) and Y rings (vertical). Each node consists of a processing element (core), a sufficiently sized send/receive buffer and a lightweight router. Data exchange takes place via *flits* that are sent from one node to another over the NoC. Because the information where a flit is to be sent is included seperately, there is no need of a head flit. Flits are 32 bits wide and are forwarded instantly without buffering. They first take the X direction and then the Y direction (xy-routing). When flits change their direction from X to Y, they are stored in a so-called corner buffer until it is the right time to leave the node. The right time to leave and arrive at buffers is determined by a schedule, see the following Section 2.3.

The goal of our implementation is to abstract communication in a parallel program: instead of making a detailed WCTT analysis for every program again, the already known WCET estimates of MPI collectives may be used. The WCET estimate of a program can then be determined by adding the WCET estimates of MPI operations to the WCET estimates of sequential parts. Because all nodes execute the same sequential code and we take its WCET estimate, it can be assumed that they are all finished when reaching a MPI operation. However, there are some restrictions to enable our implementation to stay general, e. g. no derived data types are allowed and we assume that no flit gets lost. Sometimes, more flits have to be sent at collective operations than with direct communication. One example are acknowledgement flits, which contain no real data, but only the information that a communication partner is ready.

2.3 Scheduling

TDM means that shared resources are available for each requester for a fixed time interval. Each of them has its own time slot – these are ordered in a way that no conflicts can occur. At a NoC this means that for each pair of senders and receivers it is clear when their flit is at which location in the NoC – ensuring that no other node will place a flit there at the same time. This enables estimating a WCTT – first, each participant has to wait until its time slot is available, then the flit can be transmitted. The time intervals and their order form one *round* of a TDM schedule.

There are two types of schedules: custom and generic schedules. Custom schedules are computed for the specific configuration of applications on a core. They feature a good (worst-case) performance, but each time something is changed, they have to be recomputed since a change may result in a conflict. Generic schedules are application-independent – they describe a regular pattern: when is each node allowed to send flits? How many? Which nodes are allowed for receiving these flits?

The (worst-case) performance of generic schedules is worse than that of custom schedules. However, it is possible to give general statements and changes are easier to handle than with custom schedules. In our case, we make general statements about the timing behaviour of MPI collective operations. Our timing estimation applies for any application utilising this collective operation, presuming that the same NoC and schedule is used.

Schoeberl et al. propose a generic All-To-All schedule (AA) [13]: within one period of the schedule, each node is allowed to send flits to any other node. However, each node is allowed to send at most one flit to the same node. In the worst-case this means that all nodes send one flit to each other node. Mische et al. propose a One-To-One schedule (11) amongst others [11]: each node is allowed to send and receive at most one flit in one period. This is much stricter than All-To-All, but results in shorter periods, when not all nodes participate. Furthermore, the worst-case is the same as the average case: each node sends and receives one flit. Sending several flits takes several rounds. A detailed comparison of different generic schedules for MPI collective operations takes place in [15].

3 Timing Analysis of MPI Collectives

3.1 Setting for the Analysis

Generally, our approach is platform-independent. However, to get concrete numbers, we make a timing analysis on a custom platform. It is composed of 16 nodes connected via the PaterNoster NoC and arranged as a 4x4 unidirectional torus. The NoC has been already described in Section 2.2. Each node consists of a simple core with ARM instruction set, 5 stage pipeline, no caches and 10 cycles memory access latency to the local memory. We apply a 1:1 mapping meaning that each core executes one thread.

We utilise the static WCET analysis tool OTAWA [3] for the sequential program parts. 3 cycles are assumed for the assembler instructions for sending and receiving a flit. Another 4 cycles are assumed for the time between execution of the assembler instructions and availability of the flits at the NoC buffer (and vice versa)[2]. This time can be changed at any time by adjusting the parameter t_{Buf} at the equations in the rest of this paper. For communication between nodes we considered two schedules: All-To-All as described by Schoeberl et al. [13] or One-To-One as described by Mische et al. [11]. Their WCTTs are described by the following two equations, where n is the dimension of the NoC (4 in our case), f is the number of flits to be transmitted and χ is the number of participating nodes (excluding the master node). Details can be found in [11, 15]:

$$WCTT_{All-To-All} = \frac{n^2 \cdot (n+1)}{2} \cdot f + \frac{n^2}{2} + 2n \tag{1}$$

$$WCTT_{One-To-One} = n \cdot \chi \cdot f + 2n \tag{2}$$

[2] This time is very short because our group develops hardware support for fast message passing to substitute shared memory synchronisation.

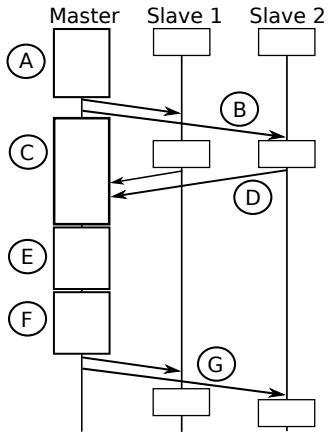


Figure 1 MPI_Allreduce with one master and two slave nodes.

Applying these equations with parameters later needed in this paper gives following numbers: When 1 flit is to be transmitted in a 4x4 NoC with All-To-All schedule, we get a WCTT of 56 cycles. With the same schedule, the WCTT is 136 cycles for 3 flits, 616 cycles for 15 flits or 14 056 cycles for 351 flits. Utilising the One-To-One schedule, 1 flit can be transmitted to or from 2 participating nodes with a WCTT of 16 cycles or 351 flits in 2 816 cycles. When χ and f are both 15 at the One-To-One schedule, the WCTT is 908 cycles, while it would be 44 cycles when they are both 3. These numbers will be used later in our analysis.

3.2 MPI_Allreduce

MPI_Allreduce and its variations are defined as "global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group" [4]. Thereby, all participating nodes send their values to the master node, which combines them with the given operation. In the benchmark example in Section 4, this operation is summing up the values. When the master node has finished collecting and totalising, it sends the result to all participating nodes.

3.2.1 MPI_Allreduce: Structure

The implementation of MPI_Allreduce is structured as follows (letters correspond to Figure 1):

(A) First, there is a short initialisation phase.
(B) Then, the master node sends an acknowledgement flit to all participating nodes.
(C) While the flits are on their way, the master node initialises some data structures preparing receiving of the values and the operation to be performed on them. At the slave nodes, there is a small sequential code after receiving the acknowledgement flit.
(D) Now, the slave nodes send their values to the master node. When more than one value should be sent, slave nodes go on sending without waiting for further acknowledgement flits from the master node.
(E) The master node gathers the values sent from the slave nodes and collects them in an array. Afterwards, the master node copies its own values to the array.
(F) Then, the collective operation is applied on the collected values (e. g. summing up values).
(G) Finally, the result of the operation is broadcasted to all participating nodes.

Table 1 Execution steps and their estimated WCET contribution to MPI_Allreduce.

Step	Estimated WCET contribution
A	73
B	12χ (local processing of broadcast)
C,D	$\max(23 + 6n^2 + 11\chi, 2 \cdot (t_{transm,\chi} + t_{Buffer}) + 24)$
D	$(f-1) \cdot \max(35\chi, t_{transm,\chi})$ (when there is more than one value)
E	$35\chi + 15 + 32f$ (processing and copying own values)
F	$42 + (\chi + 1) \cdot (94 + 23f)$ (arithmetic op.) or $42 + (\chi + 1) \cdot 41$ (bitwise op.)
G	$14 + f \cdot (11 + 12\chi) + f \cdot t_{transm,\chi} + t_{Buffer} + 35$

3.2.2 MPI_Allreduce: Timing Analysis

There is one prerequisite for the timing analysis: data structures have to be allocated statically. Since MPI_Allreduce gets the data structures handed over as calling parameter, this is left to the program.

Table 1 illustrates the execution steps of MPI_Allreduce and their contribution to the WCET estimate: After the initialisation, the broadcast of an acknowledgement flit is prepared at the master node (A). This is sent out and processed by the slave nodes (B), who reply with the (first) value to be sent (D). Meanwhile, the master node prepares data structures needed for the receiving and the collective operation (C). When C is finished before D, it has to wait for D and vice versa. Therefore, the maximum of C and D has to be taken into account for the WCET estimate.

In the case that more than one value is to be sent, the maximum of the time to receive flits (processing received values at the master node and store them in an array) and the time to transmit flits has to be determined (step D). At step E, the last received values are stored in the array and the values from the master node are appended. Afterwards, an arithmetic operation (e.g. SUM, MIN, MAX) or a bitwise operation (e.g. AND, OR, XOR) is applied on the collected values (F). Finally, broadcasting of the results is prepared and performed, followed by postprocessing in step G.

The variables in Table 1 were already described in Section 3.1. Additionally, two types of transportation times exist: $t_{transm.,\chi}$ is the time to get χ flits transported from all nodes to one node or vice versa, while t_{Buf} is the time flits need to pass from the NoC to the pipeline and vice versa. We assume 4 cycles to move from the sender's core to the NoC router and 4 cycles to move from the receiver's NoC router into the pipeline, which are together 8 cycles.

Alltogether, the WCET estimate of MPI_Allreduce can be rewritten as the following equation (the elements from Table 1 are summarised and transformed):

$$\begin{aligned} WCET_{AR}(f,\chi) = {} & 273 + 35f\chi + \max(23 + 6n^2 + 11\chi, 24 + 2(t_{transm,\chi} + t_{Buf})) \\ & + 141\chi + (f-1)\max(35\chi, t_{transm,\chi}) + (66 + t_{transm,\chi})f + t_{Buf} \end{aligned} \tag{3}$$

Utilising the numbers from Section 3.1, WCET estimates for several scenarios can be computed. For example, the WCET estimate of MPI_Allreduce is 6 698 cycles for 2 flits to be transmitted to or from 15 nodes when the All-To-All schedule is used, while it is 8 158 cycles with the One-To-One schedule. When f is 351 and χ is 3, All-To-All's WCET estimate is 156 373 cycles while One-To-One's WCET estimate is 113 073 cycles. Due to traversal times which are influenced by group sizes, One-To-One scheduling is better than All-To-All scheduling in the second case.

Table 2 Execution steps and their estimated WCET contribution to MPI_Sendrecv.

Step	Estimated WCET contribution
Initialisation	20
acknowledgement sender-receiver	$\max(5, t_{transm,1} + t_{Buf})$
time between acknowledgements	7
acknowledgement receiver-sender	$\max(5, t_{transm,1} + t_{Buf})$ (only if sender != receiver)
initialisation for sendrecv-loop	15
sendrecv-loop	$15 + \max(f \cdot 32, t_{transm,f}) + t_{Buf}$
postprocessing, finish function	51

3.3 Analysis of MPI_Sendrecv

Analogous to MPI_Allreduce, we analyse the operation MPI_Sendrecv. Its purpose is to send and receive messages at the same time. The communication partners for sending and receiving do not need to be the same. Table 2 illustrates the parts of MPI_Sendrecv and shows their calculated WCET estimates.

f is the maximum of the number of flits to be sent and received. $t_{transm,f}$ is the time needed to transmit f flits through the NoC, while $t_{transm,1}$ is the time needed to transmit 1 flit through the NoC.

The process of MPI_Sendrecv works as follows: After the initialisation, the sender sends an acknowledgement flit to the receiver, who waits for it. When the communication partners for sending and receiving are different, the other core also acknowledges the communication. After finishing acknowledgements, the loop for sending and receiving data is first prepared, then executed. Finally, some postprocessing takes place before the function is finished.

Since transmission and buffer times are expected to be greater than five, the equation may be written as shown in Equation (4):

$$WCET_{SR}(f) = 108 + 2 \cdot (t_{transm,1} + t_{Buf}) + \max(f \cdot 32, t_{transm,f}) + t_{Buf} \tag{4}$$

Again, the numbers from Section 3.1 can be used in this equation. For sending and receiving 351 values to/from two different nodes, the WCET estimate of MPI_Sendrecv is 14 300 with the All-To-All schedule and 11 396 with the One-To-One schedule.

3.4 Differences at varying configurations

While sequential parts always remain equal, the impact of communication parts changes with the size of the NoC: With increasing NoC size, communication times increase, too. Furthermore, the impact of the schedule also increases.

On the other side, a timing anomaly can occur at a small NoC: when the transmission is faster than new flits are provided by the core, it might have to be assumed that a flit misses a round of the schedule and has to wait until the next round is carried out (each round one flit can be transported). This leads to a "step" at the admission time of the flit, which could cause disadvantageous configurations of small NoCs being slower than a little bit larger NoCs.

Attention has to be paid at the receive buffer: a core is stalled when the send buffer is full, which means that everything still works fine because the WCET is driven by the WCTT. However, all schedules rely on free capacity at the receive buffer. When it is full, receiving of flits does not follow the schedule anymore and blows up the WCET. It could also mean that

estimating a WCET is not possible anymore. At our analysis, we assume that buffers are large enough.

4 Case study: Timing Analysis of the CG benchmark

Utilising the results from the previous Section, we analyse the conjugate gradient (CG) benchmark, which is taken from the NAS Parallel Benchmark Suite[3] [1, 2]. It is described as following: "a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, using unstructured matrix-vector multiplication" [1]. The NAS Parallel Benchmarks were developed for highly parallel systems. Therefore, they seem to be good benchmarks for embedded real-time multicores with distributed memory.

For the analysis, we decided to analyse a class S CG benchmark, which is the smallest class: The matrix size is $1400 \cdot 1400$, there are 7 nonzero values per row and 15 main benchmark iterations take place[4]. In the benchmark, the matrix is divided into equal sized blocks and each block is assigned to one core for computation.

Several changes had to be done for analysis: first, the benchmark had to be ported from FORTRAN 77 to C99. Then, we had to ensure that no data structures are allocated dynamically. Since OTAWA's support for floating point is not yet available for the ARM instruction set, we used integers instead of doubles[5].

The structure and estimated WCETs' contributions of the benchmark are illustrated in Table 3. Thereby, we did not analyse the initialisation of the benchmark (which is not included in the table), but the parts which are intended for benchmarking a system. This means the steps 1 to 18 have to be executed: After a sequential part at the beginning and one execution of MPI_Allreduce, a `for` loop is executed, which is iterated 15 times (estimated WCETs in the third column are for one iteration, those in the last column for 15 iterations). Afterwards, four sequential parts and communication parts alternate, before the end of the main iteration loop is reached. In the right column of Table 3 it is shown how much each step totally contributes to Equation 5.

Altogether, CG benchmark's estimated WCET can be summarised as follows:

$$\begin{aligned} WCET_{cg} = 1\,896\,959 + WCET_{AR}(2,15) + 17 \cdot WCET_{AR}(1,3) \\ + 16 \cdot (WCET_{AR}(351,3) + WCET_{SR}(351)) \end{aligned} \tag{5}$$

1 896 959 is the sum of the sequential parts from Table 3, where the for loop was respected with 15 iterations. Most of the variables needed to get a total WCET estimate were already computed throughout the previous Section 3. The only missing number is $WCET_{AR}(1,3)$, which is 1 323 with the All-To-All schedule and 1 071 with the One-To-One schedule. The numbers can be placed in Equation (5) and lead to following results:

$$WCET_{cg,AA} = 1\,896\,959 + 6\,698 + 17 \cdot 1\,323 + 16 \cdot (156\,373 + 14\,300) = 4\,656\,916 \tag{6}$$

$$WCET_{cg,11} = 1\,896\,959 + 8\,158 + 17 \cdot 1\,071 + 16 \cdot (113\,073 + 11\,396) = 3\,914\,828 \tag{7}$$

[3] `http://www.nas.nasa.gov/Software/NPB/`.
[4] We only analyse 1 iteration – the result may be multiplied by 15 to get the final result.
[5] This leads to loss of precision, but all computational steps keep the same. We do not focus on analysing all sequential instructions precisely, but to demonstrate the possibility of analysing parallel program structures.

Table 3 Parts of the main iteration loop and their estimated WCET contribution.

Step	Description	Est. WCET contrib.	Contrib. to Equation 5
1	Start of main iteration loop	15 929	15 929
2	MPI_Allreduce	$WCET_{AR}(1,3)$	$WCET_{AR}(1,3)$
3	**Begin of for loop (15 iterations)**	100 490	$15 \cdot 100\,490 = 1\,507\,350$
4	MPI_Allreduce	$WCET_{AR}(351,3)$	$15 \cdot WCET_{AR}(351,3)$
5	Sequential part	2 480	$15 \cdot 2\,480 = 37\,200$
6	MPI_Sendrecv	$WCET_{SR}(351)$	$15 \cdot WCET_{SR}(351)$
7	Sequential part	10 749	$15 \cdot 10\,749 = 161\,235$
8	MPI_Allreduce	$WCET_{AR}(1,3)$	$15 \cdot WCET_{AR}(1,3)$
9	**End of for loop (15 iterations)**	3 792	$15 \cdot 3\,792 = 56\,880$
10	Sequential part	100 513	100 513
11	MPI_Allreduce	$WCET_{AR}(351,3)$	$WCET_{AR}(351,3)$
12	Sequential part	2 480	2 480
13	MPI_Sendrecv	$WCET_{SR}(351)$	$WCET_{SR}(351)$
14	Sequential part	3 180	3 180
15	MPI_Allreduce	$WCET_{AR}(1,3)$	$WCET_{AR}(1,3)$
16	Sequential part	8 142	8 142
17	MPI_Allreduce	$WCET_{AR}(2,15)$	$WCET_{AR}(2,15)$
18	End of main iteration loop	4 050	4 050
odd	Sum of sequential parts		1 896 959

With the All-To-All schedule, one main iteration loop of the CG benchmark has an overall estimated WCET of 4 656 916 cycles, while it is 3 914 828 with the One-To-One schedule. The result of the One-To-One schedule is better than with the All-To-All schedule, because its periods are shorter when only a part of the nodes participates at communication with one node. However, for large groups the All-To-All schedule outperforms the One-To-One schedule.

Furthermore, it can be seen that the communication times are very large compared to the sequential program parts. This is caused by the communication intensive program structure, mainly influenced by the operations where 351 values are exchanged. These operations are executed 16 times.

5 Conclusion and Outlook

We presented our idea that collective (MPI) operations are beneficial for timing analysis of parallel distributed memory platforms with message passing communication. Timing analysis is performed of the two MPI collective operations MPI_Allreduce and MPI_Sendrecv for a platform with simple ARM cores connected via the PaterNoster NoC. Afterwards, we used these results to make a timing analysis of the CG benchmark, which does a lot of inter-core communication to move data and coordinate computation.

Maybe there are better implementations exhibiting lower WCET bounds for the MPI collectives. With this paper, we made the first step to show the feasability that MPI collectives could be one way to enable timing analysis for parallel platforms. Our next step will be to tighten the WCET estimate with (i) an improved MPI implementation, (ii) better workload distribution within MPI collectives and (iii) optimised hardware support. Furthermore, we see the need to implement and analyse more collectives, as well as making the results open source at `https://github.com/unia-sik`.

References

1 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. doi:10.1177/109434209100500306.

2 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing'91, pages 158–165, New York, NY, USA, 1991. ACM. doi:10.1145/125826.125925.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *LNCS*, pages 35–46. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-16256-5_6.

4 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015.

5 M. Frieb, R. Jahr, H. Ozaktas, A. Hugl, H. Regler, and T. Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *International Journal for Parallel Programming*, 2016. doi:10.1007/s10766-016-0432-7.

6 A. Kanevsky, A. Skjellum, and A. Rounbehler. MPI/RT – an emerging standard for high-performance real-time systems. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 157–166, 1998. doi:10.1109/HICSS.1998.656130.

7 E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. doi:10.1109/TVLSI.2015.2405614.

8 B. Lisper. Towards Parallel Programming Models for Predictability. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23, pages 48–58, Dagstuhl, Germany, 2012. doi:10.4230/OASIcs.WCET.2012.48.

9 S. Metzlaff, J. Mische, and T. Ungerer. A real-time capable many-core model. In *32nd IEEE Real-Time Systems Symposium: WiP Session*, pages 21–24, Vienna, Austria, 2011. URL: http://www.cs.wayne.edu/~fishern/Meetings/wip-rtss2011/WiP-RTSS-2011-Proceedings-Post.pdf.

10 J. Mische and T. Ungerer. Low power flitwise routing in an unidirectional torus with minimal buffering. In *Fifth International Workshop on Network on Chip Architectures*, NoCArc'12, pages 63–68, New York, NY, USA, 2012. ACM. doi:10.1145/2401716.2401730.

11 J. Mische and T. Ungerer. Guaranteed service independent of the task placement in nocs with torus topology. In *22nd International Conference on Real-Time Networks and Systems*, RTNS'14, pages 151:151–151:160, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659804.

12 C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 90–100, Dagstuhl, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.90.

13 M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Sixth IEEE/ACM In-*

ternational Symposium on Networks on Chip (NoCS), pages 152–160, May 2012. doi:10.1109/NOCS.2012.25.

14 A. Skjellum, A. Kanevsky, Y. Dandass, J. Watts, S. Paavola, D. Cottel, G. Henley, L. S. Hebert, Z. Cui, and A. Rounbehler. The Real-Time Message Passing Interface Standard (MPI/RT-1.1). *Concurrency and Computation: Practice and Experience*, 16(S1):i–322, 2004. doi:10.1002/cpe.744.

15 A. Stegmeier, M. Frieb, J. Mische, and T. Ungerer. WCTT bounds for MPI Collectives in the Paternoster NoC. In *14th International Workshop on Real-Time Networks (RTN)*, Toulouse, France, 2016.

16 R. B. Sørensen, W. Puffitsch, M. Schoeberl, and J. Sparsø. Message passing on a time-predictable multicore processor. In *IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 51–59, April 2015. doi:10.1109/ISORC.2015.15.

Parallel Real-Time Tasks, as Viewed by WCET Analysis and Task Scheduling Approaches

Christine Rochange

University of Toulouse, Toulouse, France
rochange@irit.fr

Abstract

With the advent of multi-core platforms, research in the field of hard real-time has recently considered parallel software, from the perspective of both worst-case execution time (WCET) and task schedulability (or worst-case response time, WCRT) analyses. These two areas consider task models that are not completely identical and sometimes make different assumptions. This paper draws a brief overview of the state of the art in the timing analysis of parallel tasks and tries to identify points of convergence and divergence between the existing approaches.

1998 ACM Subject Classification D.2.4 Software/Program Verification, C.3 [Special-Purpose and Application-based Systems] Real-Time and Embedded Systems

Keywords and phrases multicore, parallel tasks, worst-case execution time analysis, schedulability and worst-case response time analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2016.11

1 Introduction

After many years of improved single-core processor performance thanks to micro-architectural innovations, a ceiling has been reached due to three limitations: (i) the memory wall (the gap between processor and memory performance is still growing and can only partially be hidden using cache hierarchies), (ii) the instruction-level parallelism wall (finding enough parallelism in a single instruction stream to keep ever more intra-core resources busy becomes challenging) and (iii) the power wall (power consumption exponentially increases with the operating frequency, which comes with thermal dissipation issues). For all these reasons, the design of microprocessors has turned to multicore architectures that integrate several cores on a single chip and share some resources (such as the main memory and the interconnection network) among cores.

This trend, that first concerned servers and desktop systems, spreads now to embedded systems: they exhibit growing performance requirements, mainly to implement new functionalities (e.g. better control of gas emissions in automotive engines), but are subject to drastic constraints on power consumption and thermal dissipation.

Critical systems may not be ready yet to take the step because of the new challenges raised by these multicore architectures. But they will have to in the medium term because they also have increasing computing power needs and because off-the-shelves components likely to meet these needs will all be multicore.

However, if a multicore processor makes it trivially possible to increase performance by allowing higher task rates (several tasks can be run in parallel on different cores), it does not reduce the execution time of a particular task. In that sense, it does not help tasks to meet their deadline. To shorten execution times, parallelising task codes is needed.

It is likely that not all real-time applications can be parallelised because some of them must typically execute a series of actions in sequence. But we can think of applications that perform image processing which generally lends itself well to parallelisation.

As soon as real-time applications are parallelised, the question of their timing analysis arises. This means estimating their worst-case execution time and their worst-case response time (which is related to scheduling decisions). These last few years, a few results have been published in these two areas. Often, contributions to WCET analysis and to schedulability analysis tend to ignore their respective constraints. On one hand, schedulability analysis assumes that tasks can be preempted, migrated from one core to another one, without impairing WCET estimations. On the other hand, WCET analysis assumes that a task will run from end to end on the same core without being interrupted. Recently though, some steps towards higher integration of these two kinds of analyses have been made in the context of single-core platforms. For example, Altmeyer et al. [1] estimate cache-related preemption delays while Zhi-Hua et al. [26] introduce a WCET-aware task mapping and cache partitioning strategy. In this paper, we discuss the few solutions that have been proposed on both sides to handle parallel tasks, and we try to identify how far they fit together.

Please note that this paper focuses on software-level interferences between the threads of a parallel task. We are aware that hardware-level interferences occur when cores share resources. This includes bandwidth sharing that might generate conflicts and incur additional latencies, but also space sharing, such as when multiple cores share a single L2 cache. Although such hardware-related interferences may strongly impact worst-case execution times, we want to keep the focus of this paper on software-level interferences. For this reason, we deliberately ignore hardware-level conflicts, assuming that either they cannot occur (e.g. partitioning strategies are used [23]) or they are accounted for in WCET estimations [5, 3]. Almost all the approaches considered in this paper make a similar assumption.

The paper is organised as follows. In Section 2, we give an overview of parallel programming that support the parallel task models considered in WCET and WCRT analyses. Section 3 deals with scheduling approaches for parallel real-time tasks to be run on multicore platforms. Worst-case execution time analysis techniques for parallel codes are overviewed in Section 4. Section 5 discusses the compatibility of WCET and WCRT analyses and Section 6 summarises the paper.

2 Parallel Programming

The goal of parallel programming is to allow several computations within the same application to be performed simultaneously. To do so, a task is split into subtasks, technically processes or threads. In the following, we only refer to threads for the sake of simplicity. Generally, threads belonging to the same application share data and thus have to communicate in one way or another. Additionally, they sometimes have to synchronise their respective progress to ensure the correctness of the final result.

When the same computation must be done on each element of a set of data, a way to split a task is to split the set of data into subsets and to have each of them processed by a different thread. For example, in matrix-based algorithms, each thread may compute one column of the result matrix. Often, data parallelism leads to parallelising a loop in such a way that each thread executes one part of the loop iterations.

Task parallelism is another kind of parallelism where the application includes several more-or-less independent computations that may exhibit precedence constraints. In such a case, a thread is created for each computation or sequence of computations.

Of course, a hybrid pattern is possible, where the application contains several computations that can be run simultaneously and some of these computations exhibit data parallelism. Software pipelining is a particular parallel programming pattern: each piece of data must undergo the same sequence of computations and one thread is created for each computation (pipeline stage). All the threads execute in parallel and data flow from one thread to the next one.

According to the memory organisation (either a shared memory with a common address space or a distributed memory with local address spaces), the way shared data can be exchanged among threads differs. In the case of a common address space, threads simply access shared variables in memory. However, precautions must be taken to enforce the integrity of data. For example, read-modify-write sequences must be performed atomically to avoid inconsistencies. In the case of a distributed memory, threads communicate though message passing: the thread that hosts the piece of data in its local memory provides it by executing a `send()` primitive, while the thread that requires the data executes a `receive()` primitive.

Two kinds of synchronisations are typically used in parallel programs. Some of them ensure a coherent progress of threads, e.g. that one thread cannot execute beyond a given point X until another thread has reached a given point Y. Mechanisms that implement such progress synchronisations include barriers and conditions. Other synchronisations enforce atomicity for a region of the code, called a critical section, by ensuring that no other thread can execute the same region of code at the same time. This is referred to as mutual exclusion and locks are a simple way to implement it.

3 Scheduling Parallel Tasks

In the field of real-time multiprocessor scheduling, earlier works considered independent sequential tasks [4]. In this paper, we focus instead on scheduling parallel tasks. In this area, a parallel task is defined as a collection of subtasks that can be run in parallel as long as their precedence constraints are fulfilled. Each release of the task generates as many jobs as subtasks and the task instance is considered completed when every related job has been completed. We will first present common parallel task models and the underlying assumptions, then we will provide a brief overview of real-time scheduling approaches for such parallel tasks.

3.1 Parallel Task Model and Assumptions

The most general model used to represent real-time parallel tasks is a directed acyclic graph (DAG) $G = (V, E)$ that exposes subtasks (nodes, $v \in V$) and their precedence constraints (edges, $e \in E$) [2], as illustrated in Figure 1. The WCET of each node/subtask is supposed to be known. The task is characterised by a relative deadline and a period, that apply to the whole DAG. When the task is released, a job is created for each subtask. Jobs are scheduled on several cores in a way that meets precedence constraints and they all must be completed before the deadline. Note that subtask dependencies mentioned in real-time scheduling literature usually denote such precedence constraints exclusively.

This model assumes that the task is mapped to n identical cores that are not shared with other tasks. However the number of cores allocated to the task might not allow running all the ready threads simultaneously.

Other works consider the fork-join model where a task is released as a master thread that creates child threads which can run simultaneously (i.e. that do not exhibit precedence

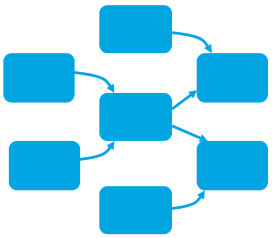


Figure 1 A parallel task defined as a set of sub-tasks subject to precedence constraints (DAG).

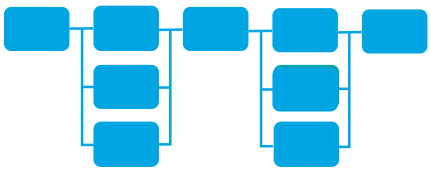


Figure 2 A parallel task following the fork-join model.

constraints) before joining [12]. This results in a sequence of sequential/parallel segments, as illustrated in Figure 2. This model assumes that all the child threads in a parallel segment execute the same code (then have the same WCET). All parallel segments have the same number of threads.

A variant of the fork-join model is the multi-frame segment model [22] shown in Figure 3. The main difference resides in the fact that parallel segments may have different numbers of threads.

Note that both the fork-join and multi-frame segment models can be expressed as DAGs.

3.2 Scheduling Approaches

There are two main kinds of approaches to scheduling real-time parallel tasks. The first one, called *task decomposition*, transforms a parallel task expressed as a DAG of subtasks into a set of independent tasks that can be handled by multiprocessor scheduling strategies for sequential tasks. The second kind of approach, referred to as *direct scheduling*, considers a parallel task as a set of subtasks that can be scheduled on different cores but must altogether meet the timing constraints of the task. These approaches are summarised below.

3.2.1 DAG Transformation

The goal of DAG transformation techniques is to convert the set of dependent subtasks into independent tasks that can then be scheduled using multiprocessor scheduling approaches for sequential tasks. Each new sequential task has its own computing requirements (WCET) but also its own release offset and its own deadline which are derived from the timing parameters of the original parallel task, taking precedence constraints into account [12, 22, 20].

3.2.2 Direct scheduling

Kato et al. [11] consider the fork-join task model (Fig. 2). They observe that threads in parallel segments should run in parallel to avoid deadlocks (or long delays) that might arise if a thread holding a lock was preempted. For this reason, they explore gang scheduling techniques in the context of real-time parallel tasks. Gang scheduling grants a task with as many cores as threads in its parallel segments. This way, all the threads run simultaneously.

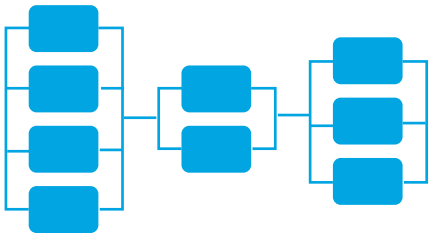


Figure 3 A parallel task following the multi-frame segment model.

Other works focus on the DAG model (Fig. 1). The federated scheduling strategy allocates a sufficient number of cores to each high-utilization parallel task and schedules its DAG nodes using a greedy algorithm [13]. Global-EDF-based approaches have also been proposed [2, 14].

4 WCET Analysis of Parallel Applications

Scheduling approaches generally summarise a task as an execution segment characterised by a worst-case execution time (often denoted as C_i), together with other attributes such as a period, a deadline or a release time. WCET analysis sees it as a control flow graph (CFG), that expresses theoretically possible execution paths and is built from the executable code. Part of the analysis exploits knowledge of the code semantics.

Research on WCET analysis started more than 20 years ago and investigates two main branches: flow analysis, that computes loop bounds and infeasible paths [7, 16, 10], and low-level analysis, that determines the local WCET of basic blocks (indivisible sequences of instructions) taking the characteristics of the underlying hardware architecture into account [24, 21, 19].

Recently, several contributions have been made to the WCET analysis of parallel tasks. Some of them consider a distributed memory organisation while other ones assume a shared memory and focus on synchronisations. All these works assume that each thread of the parallel application runs on a separate core and that there are enough cores to run all the threads simultaneously. They further consider that all threads are released and start executing at the same time. Note that they all rely on static WCET analysis techniques. However, some of their principles have been adapted to measurement-based analysis with the RapiTime tool [25].

4.1 Communicating Tasks

Potop-Butucaru and Puaut [18] consider two distinct control loops (an infinite loop that typically repeats the following cycle: read sensors, process, write actuators) running on separate cores. They assume that these loops share some data and communicate by message passing.

Their analysis aims at determining the WCET of the longest loop body, taking into account delays due to blocking message passing primitives. To achieve this, they first perform flow and low-level analyses on the control flow graphs of both loops. Then they merge the CFGs by adding edges (dashed red edges in Figure 4) to express precedence between CFG nodes (a thread calling a `receive()` primitive is stalled until the other thread executes the corresponding `send()` primitive). Finally, they compute the worst-case path in the joint CFG using the commonly used IPET method [15].

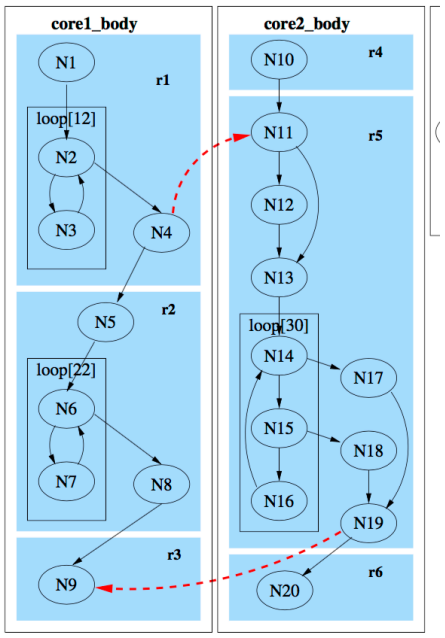


Figure 4 Joint CFG of parallel control loops (Figure taken from [18]).

This approach fits well message-passing asymmetric communications but cannot be applied to symmetric communication/synchronisation schemes, when the interleaving of threads is decided dynamically (e.g. when the order in which threads enter a critical section depends on which thread reaches the critical section first).

4.2 Parallel Tasks

Contributions in this category consider parallel tasks using shared memory and focus on delays incurred by synchronisations (barriers, conditions, locks).

Gustavsson et al. [8] apply model-checking techniques to a system of timed automata to compute the WCET of a parallel task composed of synchronising threads running on a multicore processor. Both the behaviour of the hardware executing the code and the conflicts between threads to acquire locks are modeled with timed automata that are then combined. Then properties such as "the clock value must be lower than x for every possible state" are verified considering several values of x selected following a binary search scheme.

Gustavsson et al. [9] introduce a shared-memory parallel programming language and a fix-point analysis able to identify all the possible thread interleavings at critical sections.

The approach proposed by Ozaktas et al. [17] sticks to the usual static WCET analysis process (flow analysis, low-level analysis and IPET method) and only modifies basic blocks local WCETs to reflect stall times at synchronisation points. The way worst-case stall times (WCST) are estimated is illustrated in Figure 5. It requires being able to compute *partial WCETs*, e.g. the WCET from one point in the code to another one.

The WCST for thread t to enter a critical section (i.e. to acquire a lock) is given by:

$$WCST^t = \sum_{0 \leq i < n, i \neq t} w_i^{cs}$$

where n is the number of parallel threads and w_i^{cs} is the worst-case execution time of thread i

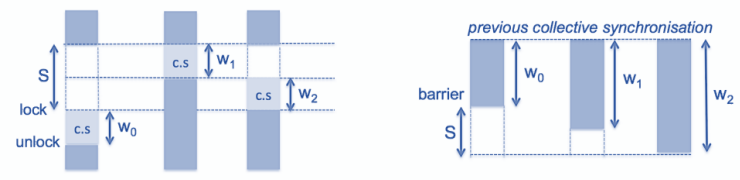


Figure 5 Computing worst-case delays at synchronisations.

in the critical section. If all the threads are identical, we get: $WCST^t = (n-1) \times w^{cs}$. This formula assumes that the lock is granted with a FIFO policy [6].

The WCST at a barrier must reflect the staggered arrival of threads at the synchronisation point. To estimate the maximum delay, we need to refer to a point in the execution where all the threads expected at the barrier formerly synchronised: this point can be another barrier or the point where threads were created since we assume that they all were created simultaneously. Then we can compute:

$$WCST^t = \max_{0 \leq i < n} (w_i - w_t)$$

where w_i is the worst-case execution time of thread i from the previous collective synchronisation to the barrier. All the synchronisation-related stall times in the task can be upper bounded this way, from partial WCET estimates. They are then added to the local WCETs of corresponding basic blocks before computing the WCET of the task, that is the WCET of its master thread.

5 How Do WCRT and WCET Analyses Fit Together?

In this section, we discuss the compatibility of WCRT and WCET analyses for parallel tasks, from three different perspectives: do they consider similar task models, do they manipulate coherent timing data and do they rely on consistent assumptions?

5.1 Different Task Models, Different Focus, Different Assumptions

Schedulability analysis typically represents a parallel task as a DAG of subtasks (see Section 3.1) while worst-case execution time analysis examine its CFG. These different representations reflect different granularity of analysis.

The main difference between the two areas is the kind of interferences that are considered among subtasks. In WCRT analysis, they are limited to precedence constraints (more precisely, other interferences are assumed to be accounted for in subtasks WCET estimations). It can happen that subtasks work on independent data and do not need to synchronise except for enforcing precedence (e.g. using condition signalling): the parallelisation process splits a sequential task into smaller scheduling units to favor a better utilisation of all the available cores. WCET analysis can support such a model by evaluating each subtask independently. However, the contributions reported in this paper give emphasis to data sharing and to synchronisation delays. This obviously makes sense for so-called *data-parallel* applications, that perform the same computation on a large set of data: often, the computation performed by one thread uses data allocated to another thread (data sharing, that may require protected access) and threads need to synchronise (e.g. at the end of each iteration of a loop) to produce correct results. Nevertheless, control loops may also exhibit data sharing and thread

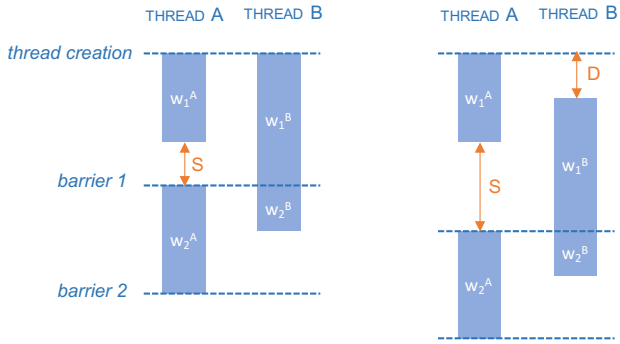


Figure 6 Simultaneous vs. staggered thread creation.

synchronisations, as observed within the parMERASA European project on parallel software from the automotive and construction machinery domains [25].

Furthermore, WCRT and WCET analyses consider different timing information. Most of the approaches in the field of WCRT/schedulability analysis require the knowledge of the WCET of each individual subtask. Worst-case execution time analysis produces instead a WCET estimation for the whole task, that is for its master thread.

Both domains make assumptions on guarantees given by the other one. WCRT analysis postulates that WCET estimations are safe regardless of scheduling decisions. On the other hand, WCET analysis assumes that all the threads belonging to the same parallel segment are created at the same time and run simultaneously on different cores (which is possible only if the number of threads does not exceed the number of available cores). This assumption is correct with gang-style scheduling strategies [11] that allocate enough cores to execute at the same time all the threads in a parallel segment. But it may not hold with the scheduling approaches that have been discussed more recently [2, 13, 14].

A staggered scheduling of threads in a parallel segment may impair the safeness of WCET bounds. Estimating the worst-case stall time at a barrier requires determining a common former synchronisation point for all the threads expected at the barrier. As shown in Section 4.2, this can be another synchronisation barrier or the point where the threads were created (more precisely, the point where they started executing). If threads are started asynchronously and in a way that is unknown at analysis time, it might be difficult to identify such collective synchronisations. This is illustrated in Figure 6. On the leftmost diagram, both threads start simultaneously: the worst-case stall time can be computed as explained in Section 4.2. On the rightmost diagram, thread B starts later than thread A. Unless an upper bound on D can be specified to the WCET analysis, the WCST (S) at barrier 1 cannot be estimated.

5.2 Towards Further Integration

Schedule-dependent WCET analysis of parallel tasks could be designed in order to to relax the need of co-scheduling all parallel threads: different WCET values could be derived for different schedules. Information provided by the scheduling analysis could help in identifying relevant schedules. For example, whether a pair of threads should be released simultaneously or not, and if not, what the release offset could be, might help in computing a correct WCET for that particular schedule. Similarly, an indication that all the threads competing for a critical section cannot run at the same time would make is possible to account for shorter stall times.

On the other hand, scheduling decisions could be taken from an enriched task model. For example, scheduling directives or constraints could be specified as conditions that the provided WCET estimations are safe. Such directives could enforce the simultaneous schedule of a given group of threads that compete for a critical section, or, on the contrary, split the group in subgroups that should not run at the same time to reduce estimated stall times. More general information on how scheduling decisions may impact WCETs could also be exploited at scheduling time to better drive these decisions.

6 Conclusion

Several recent papers in the domain of real-time systems deal with parallel tasks, either from the schedulability analysis or the worst-case execution time analysis point of view. The goal of this paper was to understand whether they all consider the same task model and make consistent assumptions, and to determine how far the different approaches could be combined. It appears that contributions to WCRT analysis mainly focus sets of subtasks with precedence constraints, while contributions to WCET analysis are more concerned with inter-thread data sharing and progress synchronisation, assuming a trivial scheduling scheme that grants a parallel task with as many cores as threads.

From this overview, it appears that both fields need to cooperate more closely. Some information about subtasks schedules might help to refine WCET estimations, or even to make them safe. Conversely, a more detailed view of subtasks (including information produced by WCET analysis) could be exposed to scheduling approaches.

Acknowledgements. The author would like to thank the organizers, the speakers and the participants of the Optimizing Real-Time Systems workshop on *Parallelization of real-time tasks*[1]: they have inspired this paper.

References

1 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 48(5), 2012.

2 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A Generalized Parallel Task Model for Recurrent Real-Time Processes. In *IEEE Real-Time Systems Symposium*, 2012.

3 Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A Unified WCET Analysis Framework for Multicore Platforms. *ACM Transactions on Embedded Computing Systems*, 13(4), 2014.

4 Robert I. Davis and Alan Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4), 2011.

5 Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, pages 31–42, 2014. doi:10.4230/OASIcs.WCET.2014.31.

[1] https://digicosme.lri.fr/tiki-print.php?page=GT+OVSTR

6 Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat. Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In *Design, Automation & Test in Europe*, 2012.

7 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *IEEE Real-Time Systems Symposium*, 2006.

8 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112, 2010. doi:10.4230/OASIcs.WCET.2010.101.

9 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, pages 38–47, 2012. doi:10.4230/OASIcs.WCET.2012.38.

10 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing switch-case code with abstract execution. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 85–94, 2015. doi:10.4230/OASIcs.WCET.2015.85.

11 Shinpei Kato and Yutaka Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *IEEE Real-Time Systems Symposium*, 2009.

12 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-Core Processors. In *IEEE Real-Time Systems Symposium*, 2010.

13 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher Gill, and Abusayeed Saifullah. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *Euromicro Conference on Real-Time Systems*, 2014.

14 Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher D. Gill, and Chenyang Lu. Global EDF Scheduling for Parallel Real-Time Tasks. *Real-Time Systems*, 51(4), 2015.

15 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *ACM/IEEE Design Automation Conference*, 1995.

16 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *IEEE Conf. on Embedded and Real-Time Computing Systems and Applications*, 2008.

17 Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET analysis of real-time parallel applications. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, pages 11–20, 2013. doi:10.4230/OASIcs.WCET.2013.11.

18 Dumitru Potop-Butucaru and Isabelle Puaut. Integrated worst-case execution time estimation of multicore applications. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, pages 21–31, 2013. doi:10.4230/OASIcs.WCET.2013.21.

19 Wolfgang Puffitsch. Persistence-Based Branch Misprediction Bounds for WCET Analysis. In *ACM Symposium on Applied Computing*, 2015.

20 Manar Qamhieh, Laurent George, and Serge Midonnet. A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems. In *Int'l Conference on Real-Time Networks and Systems*, 2014.

21 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

22 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Multi-Core Real-Time Scheduling for Generalized Parallel Task Models. *Real-Time Systems*, 49(4), 2013.

23 Vivy Suhendra and Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *45th Annual Design Automation Conference*, 2008.

24 Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, 2005. URL: `http://scidok.sulb.uni-saarland.de/volltexte/2005/466/index.html`.

25 Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel G. Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernández, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka. Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore. *ACM Trans. Embedded Comput. Syst.*, 15(3):53, 2016. URL: `http://doi.acm.org/10.1145/2910589`, `doi:10.1145/2910589`.

26 Gan Zhi-Hua and Gu Zhi-Min. WCET-Aware Task Assignment and Cache Partitioning for WCRT Minimization on Multi-core Systems. In *Int'l Symposium on Parallel Architectures, Algorithms and Programming*, 2015.

Understanding Shared Memory Bank Access Interference in Multi-Core Avionics

Andreas Löfwenmark[1] and Simin Nadjm-Tehrani[2]

1 **Dept. of Computer and Information Science, Linköping University, Linköping, Sweden**
andreas.lofwenmark@liu.se

2 **Dept. of Computer and Information Science, Linköping University, Linköping, Sweden**
simin.nadjm-tehrani@liu.se

Abstract

Deployment of multi-core platforms in safety-critical applications requires reliable estimation of worst-case response time (WCRT) for critical processes. Determination of WCRT needs to accurately estimate and measure the interferences arising from multiple processes and multiple cores. Earlier works have proposed frameworks in which CPU, shared cache, and shared memory (DRAM) interferences can be estimated using some application and platform-dependent parameters. In this work we examine a recent work in which single core equivalent (SCE) worst case execution time is used as a basis for deriving WCRT. We describe the specific requirements in an avionics context including the sharing of memory banks by multiple processes on multiple cores, and adapt the SCE framework to account for them. We present the needed adaptations to a real-time operating system to enforce the requirements, and present a methodology for validating the theoretical WCRT through measurements on the resulting platform. The work reveals that the framework indeed creates a (pessimistic) bound on the WCRT. It also discloses that the maximum interference for memory accesses does not arise when all cores share the same memory bank.

1998 ACM Subject Classification D.4.7 [Organization and Design] Real-Time Systems and Embedded Systems

Keywords and phrases multi-core, avionics, shared memory systems, WCET

Digital Object Identifier 10.4230/OASIcs.WCET.2016.12

1 Introduction

Future safety-critical avionic systems will use multi-core platforms, partly because of the more complex systems requiring more computational capacity and partly because of decreasing availability of single-core processors; but there are still challenges remaining to demonstrate the predictability needed for certification.

The memory hierarchy, and more specifically, shared caches and dynamic random access memory (DRAM) is one of the major sources of timing variability in a multi-core system [9]. Parallel accesses by cores can lead to interference and either of the resources can become saturated.

Shared caches introduce a number of problems when estimating worst-case execution time (WCET): an intra- or inter-task interference may occur when tasks on the *same* core evict either their own cache lines or another task's cache line respectively. In addition, asynchronous operating system activities can result in cache pollution. Furthermore, inter-core interference is the result of a task evicting a cache line used by a task on *another* core.

The DRAM memory system is composed of a memory controller and memory devices that store the data. The controller is a shared resource in most multi-core systems, which if accessed simultaneously from multiple cores has to somehow arbitrate the accesses and this arbitration can lead to non-determinism in the time domain. DRAM memory devices are organized into ranks containing banks. Banks contain a number of rows and each row has a number of columns. For each bank there is a row buffer that is used to store the contents of one row in the bank. To read data from memory, the row containing that data must be opened and the contents read to the row buffer and from there the column containing the data can be read. Subsequent requests to the same row can be serviced with low latency, as the row is already open. If a request requires another row to be opened, this will increase the latency as the currently open row must be closed and the data written back to the row before the new row can be opened. This will also affect the worst-case response time (WCRT) if different cores request data from different rows in the same bank.

To mitigate these effects when estimating the WCET, several methods have been proposed [9]. One approach targeting the problems outlined above is the Single Core Equivalence (SCE) framework proposed by Mancuso et al. [12]. This approach combines several of the previously proposed approaches and consists of three parts: Colored Lockdown [11] for managing the shared cache; MemGuard [24] for monitoring and limiting the number of DRAM requests; PALLOC [23] for DRAM bank partitioning. Starting from single-core WCET estimations, they are able to add interference bounds resulting from shared resource usage on a multi-core platform to minimize the effects from other cores.

For some systems it may be possible to locate the data in such a way that each core can access its own private bank(s), but in the general case there will be some sharing of data between applications and these applications may reside on different cores resulting in a use-case where shared banks is a necessity. It may also be the case that we have more cores than banks, which also will result in the necessity of sharing banks. Currently, the number of cores in a multi-core chip is growing faster than the number of banks in the DRAM [8].

In this paper we consider integrating the SCE concepts in an ARINC 653 [1] real-time operating system (RTOS) designed for avionic systems. Specifically, we study the general case of bounding the interference delay when using shared DRAM banks.

The contributions of this paper are:

- We adapt the SCE approach for WCRT estimation in avionics software by integrating assumptions valid for our context, namely cache partitioning and memory bank sharing.
- We adapt a custom RTOS to restrict memory accesses according to earlier works ([14, 5, 24, 10]).
- We present a methodology for validation of the WCRT estimates using the modified RTOS, COTS multi-core hardware, and repeatable measurements.
- We show that accessing the same bank from all cores does not necessarily represent the worst-case interference delay.

The remainder of this paper is structured as follows. Section 2 contains related work and Section 3 contains relevant background. We describe our SCE adaptation and the validation in Section 4 and Section 5 respectively. We conclude the paper in Section 6.

2 Related Work

The early work on utilizing multi-core processors for deterministic systems includes CPU scheduling. Anderson et al. [2] propose a hierarchical scheduling with different levels of

execution time estimation requirements for the different criticality levels in RTCA/DO-178 [18]. Mollison et al. [13] and Herman et al. [4] continue building on that framework, turning attention to other shared resources, such as shared last-level cache and the DRAM. Both of these shared resources have to be addressed in order to make the execution times of tasks predictable.

Several methods for handling the shared cache have been proposed. These include page coloring [11, 19] and explicit reservation [20]. In our work we will adopt the latter by using dedicated cache partitions for each application.

When cores need to access the main memory due to e.g., a cache miss, the system has to somehow arbitrate among the cores. This creates another bottleneck introducing interference. The interference[1] delay experienced by one core depends on how many memory requests the other cores issue, making it hard to estimate WCRT. One way of handling this is to specify a memory request budget for each core or a group of tasks on a core and then to monitor all memory requests and temporarily disallow access by the core if too many memory requests are performed. This will result in an upper bound of the interference delay. Inam et al. [5] use the concept of multi-resource servers, where they monitor both CPU usage and memory requests for each server. Nowotsch et al. [14] focus on extending existing estimation techniques by introducing an interference-delay analysis and a run-time monitoring mechanism that make it possible to analyze each task in isolation and then add the interference delay to account for the shared resources. A similar approach is presented by Fernandez et al. [3], who introduce resource usage signatures and templates to abstract the contention caused and experienced by tasks on different cores. These signatures and templates are used to determine an execution time bound instead of the actual tasks. Yun et al. [24] propose MemGuard, a memory bandwidth reservation system that provides bandwidth reservation for temporal isolation and a reclamation component. None of these monitoring systems consider the effects of the memory requests of the RTOS(es) running on the cores, this was addressed and shown significant in our earlier work [10]. In this paper we will consider both application and RTOS memory requests when profiling or estimating response times.

The memory has often been regarded as a single resource (black box) and a constant time used for the access times, but the DRAM can be in one of several states affecting the time required to perform the access. The DRAM consists of banks that can be accessed in parallel; Yun et al. [23] use this to reduce the interference delay when accessing the DRAM. They implement a memory allocator (PALLOC) that reserves memory in private banks for each core. This will also eliminate any row collisions, where two cores access different rows in the same bank. Row collisions are more expensive than row hits as the currently open row has to be closed. Wu et al. [21] and Kim et al. [8] both model the memory system more realistically than as a single resource. The memory controller has one request queue for each DRAM bank. Wu et al. only consider private DRAM banks for each core. Kim et al. who also include shared banks relax this limitation. Yun et al. [22] study interference arising in COTS platforms that can generate multiple outstanding memory requests and evaluate their approach on a simulation platform. In our work we use a physical COTS platform.

There are also efforts to develop predictable memory controller hardware [15, 16]. This is of course a potential scenario in the future. In this paper we are interested in deploying our system to an available COTS hardware platform in the absence of these options.

In modern memory controllers the memory requests are not always sent to the DRAM in the order they are sent by the core. Instead, they are buffered in request buffers and issued

[1] Note that in a single core context the increase in response time due to a shared resource is referred to as blocking, but we use the term interference here to stay consistent with the recent multi-core literature.

Table 1 DRAM timing parameters.

Parameter	Value	Description
BL	8 columns	Burst Length
CL	13 cycles	Column Access Strobe Latency
WL	9 cycles	Write Latency
t_{RCD}	13 cycles	Activate to read/write latency
t_{RRD}	5 cycles	Activate to activate interval
t_{RP}	13 cycles	Precharge to activate interval
t_{FAW}	26 cycles	Window for four activates
t_{WTR}	7 cycles	Write to read interval
t_{WR}	14 cycles	Data to precharge min interval
t_{CK}	1 ns	DRAM clock cycle time

to the DRAM in the order specified by a memory scheduler. The policy often used today, is the First-Ready First-Come First-Served (FR-FCFS) policy. This policy prioritizes requests to already open rows before closed rows in order to minimize row conflicts.

3 Background

In this section, we review the basic concepts relevant to this work.

3.1 DRAM Controller

A DRAM controller sends a number of commands: precharge (PRE) to close an open row; activate (ACT) to open a row; read (RD) and write (WR) to read or write data to the row buffer. The commands take time to finish and the DRAM controller must satisfy timing constraints between the commands. The JEDEC standard [6] specifies a number of requirements for JEDEC-compliant SDRAM devices as shown in Table 1.

3.2 Inter- and Intra-bank Delay

In the general case tasks share data and need to share banks. To more accurately model the worst-case memory interference delay, we build on the work by Kim et al. [8] that includes the interference delay resulting from shared banks. We briefly describe the request-driven notation used in this work to reuse in later sections.

The interference delay experienced by a core p for a memory request is given by $RD_p = RD_p^{inter} + RD_p^{intra}$, where RD_p^{intra} is the inter-bank interference delay for core p and RD_p^{inter} is the inter-bank interference delay. RD_p^{inter} is the delay due to a memory request generated by a core p is being delayed by requests from other cores due to timing effects of accessing the common command and data bus, it is given by:

$$RD_p^{inter} = \sum_{\substack{\forall q: q \neq p \ \wedge \\ shared(q,p)=\emptyset}} (L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW}) \tag{1}$$

$shared(q, p)$ is the set of DRAM banks shared between core q and core p, L_{inter}^{PRE} reflects timings of the address/command bus scheduling. L_{inter}^{ACT} is related to the minimum separation time between two activate commands sent to two different banks, and L_{inter}^{RW} is related to

the data bus contention and the bus turn-around delay as a result of the data flow direction change if a read is issued after a write or vice versa. RD_p^{intra} is a result of multiple cores accessing (different rows in) the *same* bank and is given by:

$$RD_p^{intra} = reorder(p) + \sum_{\substack{\forall q: q \neq p \ \wedge \\ shared(q,p) \neq \emptyset}} (L_{conf} + RD_q^{inter}) \tag{2}$$

where $reorder(p)$ calculates the delay from the reordering based on the number of queued row hits ($N_{reorder}$) that may be scheduled before the request under analysis and L_{conf} is a constant that represents a row-conflict in the same bank, which requires both a PRE and an ACT command to close the current row and open a new row.

We use the request-driven approach presented by Kim et al. [8] since it does not make any assumptions on the memory requests of applications running on other cores. The job-driven approach can in some situations reduce the pessimism of the delays, but it requires us to know the number of interfering memory requests from other cores and goes against the reconfiguration ideas of Integrated Modular Avionics (IMA) [17].

Based on this, Kim et al. extend the classical response time test [7] to include the memory interference delay:

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot C_j + H_i \cdot RD_p + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot H_j \cdot RD_p \tag{3}$$

where H_i denotes the maximum number of memory requests generated by task i, C_i denotes the WCET of task i when run in isolation, R_i denotes the response time of task i, and T_i denotes the minimum inter-arrival time of task i. $R_i^0 = C_i$ and the test terminates when $R_i^{k+1} = R_i^k$.

4 SCE Adaptation

In this section we outline the steps needed to implement the SCE concepts on our evaluation platform and RTOS. We assume that the applications tasks are organized in ARINC 653 partitions scheduled in a static cyclic schedule unique for each core.

Instead of implementing the cache-coloring concept, used by Mancuso et al. [12], we utilize the ability to allocate cache ways for exclusive use by a specific core. In our system (described later) the L2 cache is set up to allocate four ways to each of the four cores. This effectively limits the available cache for each core to 512 KiB and will ensure that there is no inter-core cache interference.

The memory request monitoring within the RTOS has been modified to also suspend the partition if the counted number of memory requests exceeds a specified limit during its partition window, in effect regulating the number of memory requests that can be issued from a partition. This is accomplished by using the Performance Monitor Counters (PMCs) to generate an interrupt at overflow. The budget is replenished at the start of each period. The PMC is set up to count both the requests issued by the partition and the requests issued by the RTOS itself. The sum of the memory budgets for all partitions must not exceed the total number of requests possible during a regulation period as this would saturate the DRAM controller and introduce additional delays.

Earlier Linux based bank partitioning assumes that the page size of the memory management unit (MMU) is smaller than the row size of the DRAM (i.e. 4KiB). This makes it possible to always allocate contiguous virtual memory that will map to a set of physical

memory pages belonging to the same bank. Our chosen RTOS uses a different approach, where all memory is allocated using variable page sizes during initialization. This will minimize page misses in the MMU to be handled by the RTOS, but it will also make it very difficult to implement bank partitioning as the MMU page sizes used could possibly span multiple DRAM banks. Therefore, we aim to use the SCE concepts adapted with the shared bank interference delay estimations described in Section 2.

5 Validation of the SCE Adaptation

In this section we describe the methodology for validating the adapted SCE model using the implemented SCE mechanisms (as described in Section 4) on the platform, and show the validation results. We use the four avionics related applications from previous work [10]: Nav, Mult, Cubic and Image. Without loss of generality, in this setup all partitions consist of only one process (Nav has two, but we are only interested in the highest priority one), which simplifies the response time calculations. Equation 3 is therefore reduced to $R_i = C_i + H_i \cdot RD_p$.

5.1 Methodology

We use the following method for validating the WCRT estimations:

- We estimate the WCET and count the number of memory requests for each partition in isolation.
- The worst-case response time for each partition when executing in parallel is calculated based on the (adapted) SCE formulas.
- We measure the (worst case) response time for each partition and compare with the calculated estimates.
- For critical partitions where calculations indicate a small margin to the relative deadline, we perform additional interference studies with a memory-intensive synthetic application to ensure that maximum memory bank interference is properly accounted for.

Each application is run inside one partition and deployed to different cores and they all execute in 60 Hz. We run the system in an asymmetric multiprocessing (AMP) configuration (i.e. each core has its own instance of the RTOS).

The experiments are performed on an NXP (Freescale) T4240 using only one cluster with four cores sharing the 2048 KiB L2 cache, which is partitioned to allocate four ways for each core resulting in each core having 512 KiB of L2 cache each. Without loss of generality, we have in this work disabled the reordering of requests in the DRAM controller (i.e. $N_{reorder} = 0$).

5.2 WCET Estimation

Single-core WCET can be estimated either using static analysis, by measuring the execution time of the application when running on the target hardware or by some hybrid method. In this work we use a measurement approach where we (manually) insert instrumentation points (IPOINTs) that can be used to derive an estimate of the WCET. The four applications are run in isolation on core 0 with the rest of the cores disabled. Table 2 shows the measured WCET and the number of memory requests per period for each application and also for the RTOS.

To ensure that the IPOINTs do not introduce any unintentional probe effects, we measure the execution overhead of an IPOINT and also the number of memory requests with and without IPOINTS in the applications. The maximum execution time of one IPOINT is 23 ns,

Table 2 Characterization of partitions in isolation.

Partition	Period (μs)	WCET (C) (μs)	Memory Requests (H) (Partition)	(RTOS)
Nav	16667	14	93	54
Mult	16667	16615	21740	160
Cubic	16667	9345	45	38
Image	16667	4391	560	40

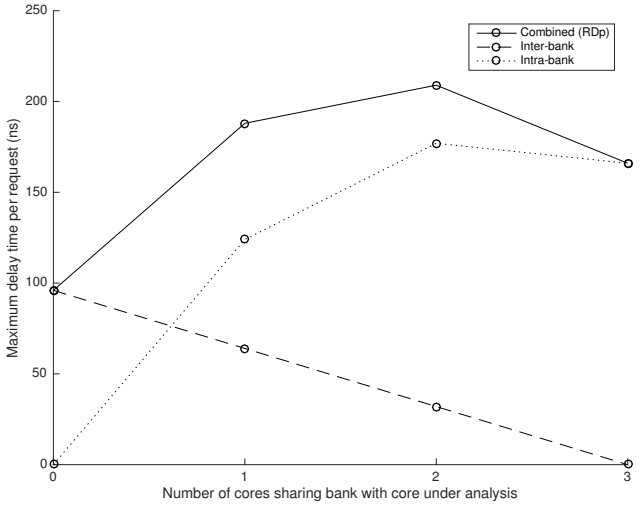


Figure 1 Theoretical interference delay when using four cores.

which gives a maximum overhead of 0.5 percent per partition window. No significant increase of memory requests is observed when using IPOINTs.

5.3 Response Time Calculations and Measurements

To calculate the response time we need the interference delay, RD_p. Using the equations in Section 2 and the DRAM timing parameters in Table 1 we calculate RD_p^{inter} and RD_p^{intra} to get RD_p. Figure 1 shows the calculated intra- and inter-bank delays as well as the combined delay (RD_p). Intuitively, one would imagine that a higher number of cores gives higher interference. However, as we can see the maximum delay does not occur when all four cores share the same bank. This is a result of intra-bank delay depending on the inter-bank delay for other cores (Equation 2). When all cores access the same bank the inter-bank delay is zero, which will result in the seen delay time drop, given an L_{conf} smaller than the $\sum RD_q^{inter}$ contributing in the case with two cores sharing bank. In the following estimates we use the maximum total interference corresponding to the highest point on the curve (209 ns).

The estimated WCRT listed in Table 3 show that the calculated response time of Nav, Cubic and Image is safely below their relative deadline, but for Mult the estimated WCRT exceeds the relative deadline. Mult performs several orders of magnitude more memory requests compared to Nav, Cubic and Image. For every partition window we use the earlier mentioned IPOINTs to record response times over a 30 second interval (1800 measurements)

Table 3 Maximum response time of partitions.

Partition	Core	Response time (R) (μs)	
		Estimated	Measured
Nav	0	45	14
Mult	1	21192	16620
Cubic	2	9362	9345
Image	3	4516	4391

Table 4 Measured maximum response time of Mult with memory intensive tasks in parallel.

Partition	Core	Response time (R) (μs)	
		No regulation	Regulation
Mult	0	17075	16654

with partition placement on cores according to column 2. The maximum measured response time is then disclosed in column 4.

This shows that when the partitions execute in parallel no partition misses deadlines though the critical application Mult has a tight margin (see periods in Table 2). We can also see that the WCRT measurements do not differ in any significant way from the WCET measurements in that table. The memory controller can service all the memory requests without being saturated. The memory access patterns of the partitions are such that they do not interfere in many instances. The estimation model assumes that all cores issue memory requests simultaneously. To ensure the measurements are not overly optimistic we perform additional measurements on Mult, whose estimated WCRT exceeds its relative deadline, in a scenario with maximal memory interference.

5.4 Studying Critical Processes Individually

When we run Mult on core 0 in parallel with a memory intensive task deployed on core 1–3 with disabled memory access regulation, Mult misses its deadline. If we turn on the memory regulation, mentioned in Section 4, for the memory intensive tasks on core 1–3 with a suitable budget we notice that Mult does not miss its deadline. This shows that *given* a suitable restriction of memory accesses by partitions running on other cores we are able to run Mult within its time constraints. So, for the critical task, the correct estimation of regulation budget of *other* tasks is essential. To measure Mult's response time in this scenario we disable the overrun detection function and perform repeated experiments where the memory budgets of the memory intensive tasks were reduced until a WCRT value below the relative deadline was found for Mult. The resulting response times (with and without regulation) can be found in Table 4.

5.5 Applying the Method to an Earlier Benchmark

To further assess the validity of the approach we also use the *Latency* and *Bandwidth* benchmarks from Yun et al. [24], adapted to our environment and RTOS, to measure the worst-case memory interference. The benchmarks are modified to enable us to direct the requests from *Bandwidth* to a specified DRAM memory bank. We run *Latency* on core 0 and *Bandwidth* on core 1–3 several times with different number of *Bandwidth* instances

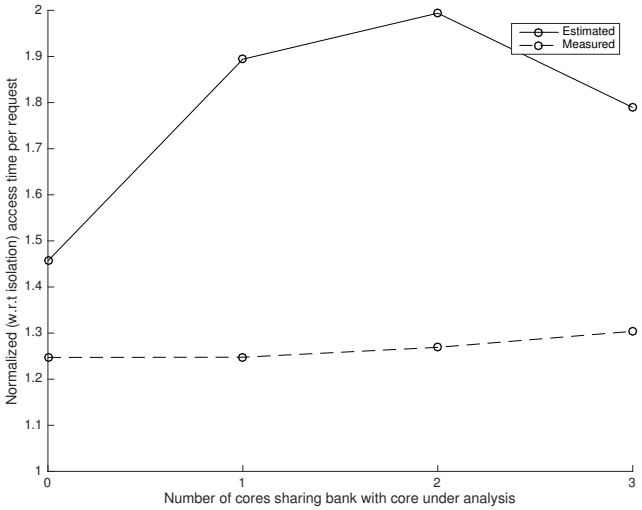


Figure 2 Comparison of measured and estimated request time.

targeting the same DRAM memory bank as *Latency*. These measurements compared to the estimations are shown in Figure 2. As we can see, the estimations are a conservative (and possibly somewhat pessimistic) approximation of the measurements.

6 Conclusion

In this paper we have presented an adaptation of the SCE framework for an avionics ARINC 653 RTOS targeting the T4240 multi-core SoC from NXP. We have relaxed the constraints of requiring private memory banks for each core and our adaptation provides an analytic upper bound on the interference delay. The implementation on the avionics platform has been used to understand and validate the revised SCE framework using both synthetic and realistic applications for avionics systems. Our work has highlighted an interesting aspect of the calculated response times as a function of the number of cores, namely that the maximum core deployment need not give maximum memory bank interference. It also justifies the use of the SCE framework as an approach to assess schedulability of critical tasks on multi-core platforms. As future work, we will examine the DRAM request reordering we turned off in this work and an improved model of the DRAM controller for more precise estimates.

References

1 Aeronautical Radio Inc (ARINC). ARINC 653: Avionics application software standard interface part 1 – required services, 2010.

2 James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proc. of Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

3 Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Resource usage templates and signatures for cots multicore processors. In *Proc. of 52nd Annual Design Automation Conference*, DAC'15, pages 155:1–155:6, New York, NY, USA, 2015. ACM. doi:10.1145/2744769.2744901.

4 J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. RTOS support for multicore mixed-criticality systems. In *Proc. of 18th IEEE Real-Time and*

Embedded Technology and Applications Symposium, pages 197–208, 2012. doi:10.1109/RTAS.2012.24.

5 R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

6 Joint Electron Device Engineering Council (JEDEC). DDR3 SDRAM Standard, 2012. URL: http://www.jedec.org/standards-documents/docs/jesd-79-3d.

7 M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986. URL: http://comjnl.oxfordjournals.org/content/29/5/390.abstract, arXiv:http://comjnl.oxfordjournals.org/content/29/5/390.full.pdf+html, doi:10.1093/comjnl/29.5.390.

8 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 145–154, 2014. doi:10.1109/RTAS.2014.6925998.

9 A. Löfwenmark and S. Nadjm-Tehrani. Challenges in future avionic systems on multi-core platforms. In *Proc. of 25th IEEE International Symposium on Software Reliability Engineering Workshops*, pages 115–119, 2014. doi:10.1109/ISSREW.2014.70.

10 A. Löfwenmark and S. Nadjm-Tehrani. Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform. In *Proc. of 26th IEEE International Symposium on Software Reliability Engineering*, pages 153–160, 2015. doi:10.1109/ISSRE.2015.7381809.

11 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. of 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

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

13 M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proc. of 10th International Conference on Computer and Information Technology*, pages 1864–1871, 2010. doi:10.1109/CIT.2010.320.

14 J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Proc. of 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014. doi:10.1109/ECRTS.2014.20.

15 M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. *IEEE Embedded Systems Letters*, 1(4):86–90, 2009. doi:10.1109/LES.2010.2041634.

16 Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *Proc. of the 7th International Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011. doi:10.1145/2039370.2039388.

17 RTCA, Inc. RTCA/DO-297, integrated modular avionics (IMA) development, guidance and certification considerations, 2005.

18 RTCA, Inc. RTCA/DO-178C, software considerations in airborne systems and equipment certification, 2012.

19 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *Proc. of 25th Euromicro Conference on Real-Time Systems*, pages 157–167, 2013. doi:10.1109/ECRTS.2013.26.

20 Jack Whitham, Neil C. Audsley, and Robert I. Davis. Explicit reservation of cache memory in a predictable, preemptive multitasking real-time system. *ACM Trans. Embed. Comput. Syst.*, 13(4s):120:1–120:25, April 2014. doi:10.1145/2523070.

21 Zheng Pei Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *Proc. of 34th Real-Time Systems Symposium*, pages 372–383, 2013. doi:10.1109/RTSS.2013.44.

22 H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Proc. of 27th Euromicro Conference on Real-Time Systems*, pages 184–195, 2015. doi:10.1109/ECRTS.2015.24.

23 Heechul Yun, R. Mancuso, Zheng-Pei Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166, 2014. doi:10.1109/RTAS.2014.6925999.

24 Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. of 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.