14th International Workshop on Worst-Case Execution Time Analysis

WCET 2014, July 18, 2014, Madrid, Spain

Edited by

Heiko Falk

Editor
Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University
Heiko.Falk@uni-ulm.de

Workshop Webpage
http://www.uni-ulm.de/wcet2014

Financial Support

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

ISBN 978-3-939897-69-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-69-9.

Publication date
July, 2014

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

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.WCET.2014.i

OASIcs – OpenAccess Series in Informatics

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

Editorial Board

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

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Welcome to WCET 2014 vii

List of Authors ix

Committee xi

Regular Papers

Principles for Value Annotation Languages
Björn Lisper 1

A Formally Verified WCET Estimation Tool
André Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut 11

On the Sustainability of the Extreme Value Theory for WCET Estimation
Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart 21

Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art
Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla 31

On Static Timing Analysis of GPU Kernels
Vesa Hirvisalo 43

A Time-Predictable Memory Network-on-Chip
Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø 53

The Challenge of Time-Predictability in Modern Many-Core Architectures
Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, José Carlos Fonseca, Marko Bertogna, Eduardo Quiñones, Roberto Vargas, and Andrea Marongiu 63

Scope-Based Method Cache Analysis
Benedikt Huber, Stefan Hepp, and Martin Schoeberl 73

Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis
Sahar Abbaspour, Alexander Jordan, and Florian Brandner 83

Identifying Relevant Parameters to Improve WCET Analysis
Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot 93

Towards Automated Generation of Time-Predictable Code
Daniel Prokesch, Benedikt Huber, and Peter Puschner 103

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk

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

Welcome to WCET 2014

It is my great pleasure to welcome you to the *14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*. This year's edition of WCET continues its tradition of being the premier forum for presentation of research results in the fields of hard real-time systems, predictability and timing analysis. One important goal of WCET is to provide a link between the timing analysis and the formal analysis, computer architecture and compiler communities. Researchers and developers in these areas are addressing many similar problems, but with different backgrounds and approaches. WCET is intended to expose researchers and developers from either area to relevant work and interesting problems in the other area and to provide a forum where they can interact.

The call for papers attracted 17 submissions from Asia, Europe and North America. The program committee accepted 10 papers that cover a variety of topics, including formal methods for WCET and value analysis, multicore challenges, timing analyses for emerging cache architectures and software approaches supporting WCET analysis. In addition, the program includes a keynote talk given by Vincent Nélis on the challenge of time-predictability in modern many-core architectures, and a report by Christine Rochange on the 2014 edition of the WCET Tool Challenge.

Putting together WCET 2014 was a team effort. First of all, I would like to thank the authors and Vincent and Christine for providing the content of the program. I would like to express my gratitude to the program committee and reviewers who worked very hard in reviewing papers and providing suggestions for their improvements.

WCET 2014 is being organized as satellite workshop of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014). I am therefore grateful to the ECRTS 2014 general chair, Juan Antonio de la Puente, his local team, and the Real-Time Technical Committee Chair of Euromicro, Gerhard Fohler, for their support. This year's WCET workshop were not possible without external support – the financial support by the EU COST Action IC1202: Timing Analysis on Code-Level (TACLe) and by the COST Office is highly appreciated.

I hope that you will find this program interesting and thought-provoking and that the workshop will provide you with a valuable opportunity to share ideas with other researchers and practitioners. These proceedings will hopefully serve as worthwhile reference for researchers in the timing analysis and real-time systems domains, enjoy reading this volume.

Heiko Falk

List of Authors

Abbaspour, Sahar 83
Abella, Jaume 31
Bertogna, Marko 63
Blazy, Sandrine 11
Bonenfant, Armelle 93
Brandner, Florian 83
Cazorla, Francisco J. 31
Chong, David Vh 53
Claraz, Denis 93
Cuenot, Philippe 93
Dufour, Guillaume 21
Fernandez, Gabriel 31
Fonseca, José Carlos 63
Hepp, Stefan 73
Hirvisalo, Vesa 43
Huber, Benedikt 73, 103
Jacquemart, Damien 21
Jordan, Alexander 83
Lisper, Björn 1
Maroneze, André 11
Marongiu, Andrea 63
Morio, Jérôme 21
Nélis, Vincent 63
Pichardie, David 11
Pinho, Luís Miguel 63
Prokesch, Daniel 103
Puaut, Isabelle 11
Puffitsch, Wolfgang 53
Puschner, Peter 103
Quiñones, Eduardo 31, 63
Rochange, Christine 31
Santinelli, Luca 21
Schoeberl, Martin 53, 73
Sotin, Pascal 93
Sparsø, Jens 53
Vardanega, Tullio 31
Vargas, Roberto 63
Yomsi, Patrick Meumeu 63
Zwirchmayr, Jakob 93

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk

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

Committee

General Chair

Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University, DE

Program Committee

- Sebastian Altmeyer
  University of Amsterdam, NL
- Guillem Bernat
  Rapita Systems, UK
- Francisco J. Cazorla
  Barcelona Supercomputing Center, ES
- Damien Hardy
  University of Rennes 1 / IRISA, FR
- Niklas Holsti
  Tidorum Ltd., FI
- Raimund Kirner
  University of Hertfordshire, UK
- Jens Knoop
  Vienna University of Technology, AT
- Kim G. Larsen
  Aalborg University, DK
- Björn Lisper
  Mälardalen University, SE
- Claire Maiza
  Grenoble INP/Verimag, FR
- Tulika Mitra
  National University of Singapore, SG
- Harini Ramaprasad
  Southern Illinois University, USA
- Jan Reineke
  Saarland University, DE
- Christine Rochange
  IRIT - Université de Toulouse, FR
- Tullio Vardanega
  University of Padua, IT

External Reviewers

- Jaume Abella
- Mihail Asavoae
- Jalil Boudjadar
- Hugues Cassé
- Andreas Engelbredt Dalsgaard
- Santanu Dash
- Andreas Gustavsson
- Sebastian Hahn
- Carles Hernandez
- Michael Jacobs
- Nilesh Karavadera
- Jin Hyun Kim
- Leonidas Kosmidis
- Abu Naser Masud
- Vu Thien Nga Nguyen
- Mads Christian Olesen
- Milos Panic
- Michael Zolda
- Jakob Zwirchmayr

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk

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

Principles for Value Annotation Languages

Björn Lisper

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

Abstract

Tools for code-level program analysis need formats to express various properties, like relevant properties of the environment where the analysed code will execute, and the analysis results. Different WCET analysis tools typically use tool-specific annotation languages for this purpose. These languages are often geared towards expressing properties that the particular tool can handle rather than being general, and mostly their semantics is only specified informally. This makes it harder for tools to communicate, as well as for users to provide relevant information to them. Here, we propose a small but general assertion language for *value constraints* including IPET flow facts, which is an important class of annotations for WCET analysis tools. We show how to express interesting properties in this language, we propose some syntactic conveniences, and we give the language a formal semantics. The language could be used directly as a tool-independent annotation language, or as a meta-language to give exact semantics to existing value annotation and flow fact formats.

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

Keywords and phrases Real-Time System, WCET Analysis, Flow Fact, Assertion

Digital Object Identifier 10.4230/OASIcs.WCET.2014.1

1 Introduction

WCET analysis tools provide means to estimate the WCET of code with increased confidence, safety and automation compared with a manual analysis. Alas, full automation is hard to attain due to a number of reasons. Some are fundamental, such as the undecidability of the WCET analysis problem, others are of more practical nature like the need to provide relevant information not present in the code, or give directives to fine-tune the analysis. Thus, WCET analysis tools have annotation languages to provide various kinds of information to the analysis. Examples are:

- annotations providing *information about the environment*, like hardware configuration, entry points of tasks, etc.,
- annotations *directing the analysis* (like selection of abstract domain, context-sensitivity, choice of internal representations, kinds of generated flow facts),
- directives how to present the analysis results,
- *value annotations* constraining the possible values of program variables in different program points, and
- *flow facts* constraining the possible program flows.

Unfortunately, the means to provide these kinds of information are not systematically developed. WCET analysis tools tend to have their own annotation languages, which may be apt to provide information for the respective tool but are not portable across tools. As for manually provided information, many of these tool-specific formats do not provide a particularly user-friendly syntax. The semantics is not always entirely clear either, due to the absence of formal definitions.

© Björn Lisper;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 1–10

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

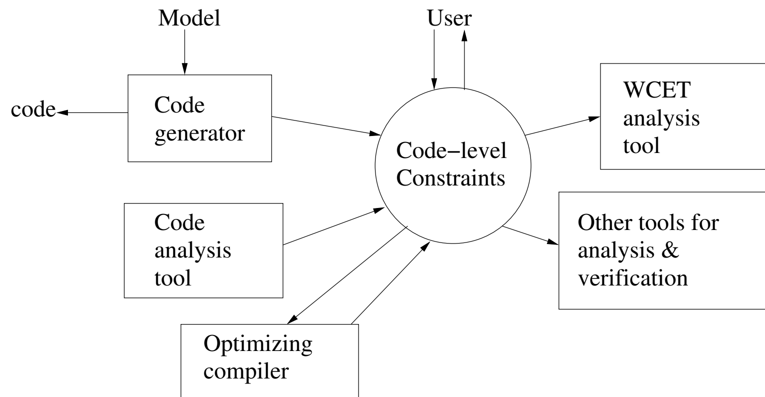


Figure 1 An ecosystem of embedded systems tools.

Furthermore, WCET analysis tools do not exist in isolation. Today one can speak of an "ecosystem" of code-level tools, as depicted in Fig. 1, including code generators for model-based development, general-purpose static code analysis tools, optimising compilers, and tools for formal verification. It is obvious that general formats for code-level constraints would be helpful for the integration of code-level tool chains.

The contribution of this paper is a step in this direction. We define a simple but general core language for *value constraints* from first principles. Such constraints provide information about reachable states, much like assertions in Floyd-Hoare logic [7, 8]. We take measures to make the constraint language as independent of the "host language" as possible, making it portable over a wide range of code formats. We make some suggestions for user-friendly syntax. The language can express numerical constraints on values of program variables, which is sufficient to express the value annotations and flow facts supported by current WCET tool annotation languages, but it can easily be extended to express constraints on other data types. We give the language a formal semantics, making minimal assumptions on the semantics of the host language. We also prove a theorem about compositionality of assertions.

2 Some Existing Annotation Languages

We now review the annotation languages for some existing WCET analysis tools.

FFX [3] is intended to be a portable WCET annotation language for flow facts. It is supported by TuBound [12], oRange [4], Otawa [1], and CalcWCET167 [10]. FFX represents the analyzed code as a structured XML document, where annotations appear as attributes in tags representing program constructs such as loops. FFX can specify upper loop bounds, and also whether or not they are exact. It is unclear whether it can represent more general linear flow facts, which describe relations between execution counters for different parts of the code. It can represent contexts: however, these seem not to be calling contexts but rather information about the environment such as the target hardware.

The annotation language of aiT [5], called AIS [6], can describe various kinds of flow facts, such as (upper) loop bounds, infeasible paths, and general linear flow constraints on IPET execution counters. Loop bounds can be complex expressions, and may for instance refer to execution counters of outer loops. Some context-sensitivity is provided by the ability to tie flow facts to certain call sites for functions. AIS can represent flow facts both for source

code (C) and for executables: locations for execution counters are formed from file name and position in the file for source code, and are provided as numerical addresses for executables.

The WCET analysis tool SWEET [18] provides a rich set of different annotations. It can read value constraints in a certain format, constraining the values of program variables in certain program points to intervals. A common use is to specify value constraints on inputs. SWEET can also compute value constraints and export them using the same format. Furthermore SWEET provides a rich set of flow facts, including general linear constraints on execution counters. Execution counters can be local to certain execution contexts (typically loop or function bodies), and are then reset at each entry of the context. Flow facts involving such counters can be constrained to certain ranges of loop iterations. Flow facts involving global execution counters are also allowed. Flow facts can be constrained to be valid only for certain calling contexts, which are specified by explicit call strings: this provides context-sensitivity.

Bound-T [9] can use a number of different assertions. It can take value range constraints for program variables, bounds on the number of loop iterations, and function calls, "path not taken"-constraints, and different bounds on stack usage. Variables can also be asserted not to have their values changed in certain parts of the program. Assertions can be tied to program parts in an interesting way, identifying, e.g., loops by properties like that they use a certain variable, call a certain function, or whether a loop is the inner or outer loop in a loop nest. Assertions can be restricted to certain calling contexts and can thus be context-sensitive. Bounds on the number of executions can also be placed directly on instructions, using addresses or offsets.

These languages have some features in common. They can all express flow facts, of different generality. The flow facts may concern global IPET execution counters as well as local execution counters, for certain execution contexts. Some of them can also express certain kinds of value constraints. They provide varying levels of context-sensitivity. Our proposed core language aims to cover these aspects in a unified way.

3 A Wish List for a Language for Code-level State Constraints

Based on experience of WCET annotation languages as well as general language design, the following wish list on a value constraint language can be formulated:

- it should work over a wide range of code-level tools (not necessarily only for WCET analysis),
- it should work over a wide range of host languages, on different levels
- it should be general yet simple, extensible, and have few but powerful constructs,
- it should have a succinct, intuitive syntax for humans, as well as an easily machine-readable form (XML) for tools,
- it should be able to express general restrictions on flow facts, including the ability to express constraints in existing annotation languages by translation,
- it should be able to express different kinds of context sensitivity, and
- it should have a clear and simple formal semantics.

This wish list has guided the design of our core language.

4 A Starting Point: The Assertion Language of Floyd-Hoare Logic

A general, existing assertion language for constraints on program variable values is the one used in Floyd-Hoare logic [7, 8]. A good description is found in [19]. It has the following elements:

- program variables, which depend on program state,
- *auxiliary* variables, which are independent of state, and
- some sublanguage to define predicates over variables (typically consisting of boolean and arithmetic expressions, including quantifiers ($\forall$, $\exists$) over auxiliary variables, but whose elements may vary depending on what kind of assertions are to be expressed

An example of a statement in Floyd-Hoare logic is

$$\{X = i\}X := X + 1\{X = i + 1\}$$

This assertion is a triple of a *pre-condition*, a program, and a *post-condition*. X is a program variable and i an auxiliary variable. The statement expresses a relation that holds between preceding and succeeding states: for any value of i, if the value of X equals i in a state preceding the program, then it will equal $i + 1$ in the state that results after having executed the program.

Floyd-Hoare logic was originally defined for a simple, structured imperative language, and it comes with a set of inference rules, based on the syntax of the language, by which assertions can be proved deductively from sub-assertions of sub-programs. However the assertions can also be given a direct semantics in terms of state transitions, which is of interest here. See [19].

5 A Core Language for Value Constraints

Floyd-Hoare logic is defined over a high-level language where the control flow is decided entirely by the syntax. We want our core language for assertions to work over a wide variety of code formats, including low-level formats with unstructured control flow. Therefore we abstract away from the syntax of the host language, and we will make only minimal assumptions on its semantics. The abstract syntax for our assertion language is given by the following:

$$\begin{array}{lll} a & ::= & n \mid i \mid X \mid a_1 \; aop \; a_2 \\ p & ::= & true \mid false \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p \mid a_1 \; rop \; a_2 \mid \forall i.p \mid \exists i.p \mid PC = L \\ c & ::= & p_1 \rightarrow p_2 \end{array}$$

Here a stands for arithmetic expressions, and p predicates. n stands for numerical constants, i auxiliary variables, X program variables, *aop* arithmetic operators ($+$, $-$, $\ldots$), and *rop* relational operator ($<$, $=$, $\ldots$). L stands for *labels*, see below. We will freely use operators that can be derived from the core language, like implication ($\implies$).

This part defines a predicate language over arithmetic expressions that is completely standard (except for the "$PC = L$" part, which will be explained below). Like in Floyd-Hoare logic the difference between program variables and auxiliary variables is that the value of a program variable depends on the state, whereas the values of auxiliary variables are independent of the state. We allow quantification over auxiliary variables, but not over program variables.

The statements of form $p_1 \rightarrow p_2$ are the assertions in our core language, and they correspond to the triples in Floyd-Hoare logic (with p_1 as pre- and p_2 as post-condition). The meaning of $p_1 \rightarrow p_2$ is "all states that are reachable from a state satisfying p_1 must satisfy p_2". We will provide an exact definition in Section 6.

We make the following assumptions on the host language and its semantics. It has program variables that can hold values, and there are labels that identify program points. At this point we assume nothing more about labels than that they can be compared for equality.

The language has states: in each state σ, a program variable X holds a numerical value $\sigma(X)$. Furthermore there is a distinguished variable PC such that $\sigma(PC)$ is a label: thus, the state also contains the current position in the code. The semantics of a program in the host language is given by a set of state transitions $\sigma \to \sigma'$.

Our assertion language is defined over conditions on numerical expressions, but is easily extended to conditions over other data types. It should also be straightforward to give the language both a user-friendly syntax as well as a conveniently machine-readable XML format. We propose some syntactic conveniences in Section 5.2.

The language so far can express sensitivity to contexts that are conditions on the state (trough implication), but it does not have any means to express call-string contexts. We show how this can be added in Section 7.

Let us now see some examples of assertions. To make the examples more concrete we assume labels "*entry*", "*exit*" representing the entry and exit point of the host program, respectively:

- $(PC = \mathit{entry}) \to (PC = L \implies X < 17)$: for all states reachable from the start of the program, if at label L then $X < 17$;
- $(PC = \mathit{entry}) \to (PC = L \wedge 3 \leq I \leq 7 \implies X < 17)$: for all states reachable from the start of the program, if at label L, with the value of I between 3 and 7, then $X < 17$;
- $(PC = \mathit{entry} \wedge 1 \leq X \leq 10) \to (PC = \mathit{exit} \implies Y \leq 100)$: if the program is started with $1 \leq X \leq 10$ then, at exit, $Y \leq 100$;
- $(PC = L \wedge X = i) \to (PC = L' \implies X = 2 \cdot i)$: for any value of i, if the program passes L with $X = i$ then afterwards, whenever at L', $X = 2 \cdot i$;
- $\mathit{true} \to X < 32767$: a global invariant, in all states holds that $X < 32767$.

Notice how restrictions on inputs, like confining an input value to a certain range, can be expressed as arithmetic contraints in the condition defining the initial states. Also note how contexts that are restrictions to certain states can be expressed simply as antecedents in implications. Such restrictions can for instance be presence at a certain program point, or that the value of a loop counter is in a certain interval.

5.1 Labels

Labels can be basically anything that identifies program points. For high-level languages like C, labels can be explicit C labels defined in the source code, or they can be formed from file name, line number, and position on the line as in AIS. Another possibility is to use paths through the parse tree of the program as labels. For low-level code a label can be a pair (e, n) where e is a symbolic entry point and n is a numerical offset, or even a fully numerical address for a linked executable.

Our basic core language only assumes that labels can be compared for equality. Certain kinds of labels, like for instance numerical addresses, can allow a richer set of conditions to specify sets of labels.

Different kinds of labels can be *fragile* to different extent, in that they may be destroyed by recompilation or editing of the source code. Examples of fragile labels are numerical addresses in executables, and line numbers in source code. Assertions that use such labels may have to be restored frequently. While interesting, the construction of non-fragile labels is outside the scope of this paper.

5.2 Syntactic Sugar

The notation developed so far can be simplified for some common cases. For instance, it can be expected that restrictions to certain program points are frequent. Thus, using the notation "$@L$" for "$PC = L$" may help. It also seems like a common case to consider all the states that are reachable from the entry point of the program: thus on the top level, where an assertion is expected, one may allow to write p as a shorthand for $@\mathit{entry} \rightarrow p$. Some of our examples above can then be written:

- $@L \implies X < 17$ (understood, for all states reachable from the entry point)
- $@L \wedge 3 \leq I \leq 7 \implies X < 17$ (similarly)
- $(@\mathit{entry} \wedge 1 \leq X \leq 10) \rightarrow (@\mathit{exit} \implies Y \leq 100)$
- $(@L \wedge X = i) \rightarrow (@L' \implies X = 2 \cdot i)$:

5.3 IPET Execution Counters and Flow Facts

IPET execution counters are "virtual" program variables that keep track of how many times a program part has been executed. Flow facts are expressed as arithmetic value constraints on these counters. The counters are typically defined relative to some execution context, with some entry and exit points, such that they are reset each time the execution context is entered and incremented by one each time the program part in question is executed. Global execution counters are defined relative to the whole program, with the *entry* label as entry point and *exit* as the exit point. For the final IPET calculation of the WCET estimate it is the possible values of the counters at exit that are of interest.

To express IPET execution counters we introduce the unary operator "#" on labels: if L is a label, then $\#L$ is the IPET counter associated with the program point of that label. We leave open how to associate IPET counters with different execution contexts, and how to exactly specify their semantics: for now we assume that they are global, and that their semantics is given by the informal description above.

We can now express flow facts in our assertion language, as value contraints on the IPET counters. Here are some examples:

- $@\mathit{exit} \implies \#L < 100$: a simple capacity constraint;
- $@\mathit{exit} \implies \#L = 99$: an exact capacity constraint;
- $@\mathit{exit} \implies \#L_1 + \#L_2 \leq 1$: a mutual exclusivity constraint;
- $(@\mathit{entry} \wedge 1 \leq X \leq 10) \rightarrow (@\mathit{exit} \implies \#L \leq 100)$: a capacity constraint under the condition that the value of X lies in the range $[1 \ldots 10]$ at entry;
- $(@\mathit{entry} \wedge X = n) \rightarrow (@\mathit{exit} \implies \#L \leq 2 \cdot n + 1)$: a parametric capacity constraint relating the number of executions of L to the value of X at entry;
- $@\mathit{exit_local} \wedge 3 \leq \#L \leq 7 \implies \#L_{local} < 17$: for each of the iterations 3 to 7 of an outer execution context with label L, L_{local} is executed less than 17 times.

In the last example $\#L_{local}$ is supposed to be a local execution counter, which is reset each time its local execution context is entered.

As the values of IPET counters at exit from their execution contexts are of primary interest, a possible syntactic simplification is to allow the "$@\mathit{exit} \implies$" part to be implicit and add it automatically when parsing a constraint that contains an IPET counter. So. for instance, the second constraint above could then simply be written $\#L = 99$, which then is to be interpreted as $@\mathit{exit} \implies \#L = 99$, which in turn stands for $@\mathit{entry} \rightarrow (@\mathit{exit} \implies \#L = 99)$.

5.4 Time

The state could also contain time. This gives the ability to express fine-grained real-time constraints on certain pieces of code. For instance L and L' may be program points in a loop, with loop counter variable I, such that we want to specify that within each iteration L' should never be executed more than 7 time units later than L. If time is represented by the program variable T, then this constraint can be expressed as

$$(@L \wedge t = T \wedge i = I) \rightarrow (@L' \wedge i = I \implies T - t \leq 7)$$

This example makes heavy use of auxiliary variables to refer to the value of a program variable in a pre-condition from the post-condition. This is quite common. A possible syntactic convenience is to make the equalities in the pre-condition implicit, and refer to the "old" value of program variable X as $X.old$ in the post-condition. With this notation, our example becomes

$$@L \rightarrow (@L' \wedge I = I.old \implies T - T.old \leq 7)$$

6 Formal Semantics

We now give a formal semantics to our core assertion language defined in Section 5. As is standard in programming language theory, we use semantic functions. These take three arguments: a syntactic form, an *interpretation* I that maps auxiliary variables to values, and a program state σ mapping program variables to values. The definitions of the semantic function $\mathcal{A}[\![\ ]\!]$, for arithmetic expressions, and $\mathcal{B}[\![\ ]\!]$, for boolean expressions (predicates), are completely standard and are given below for completeness (cf. [19]):

$$\begin{gathered}
\mathcal{A}[\![n]\!]\, I\, \sigma = n \quad \mathcal{A}[\![i]\!]\, I\, \sigma = I(i) \quad \mathcal{A}[\![X]\!]\, I\, \sigma = \sigma(X) \\
\mathcal{A}[\![a_1\ aop\ a_2]\!]\, I\, \sigma = \mathcal{A}[\![a_1]\!]\, I\, \sigma\ aop\ \mathcal{A}[\![a_2]\!]\, I\, \sigma \\
\mathcal{B}[\![true]\!]\, I\, \sigma = true \quad \mathcal{B}[\![false]\!]\, I\, \sigma = false \quad \mathcal{B}[\![p_1 \wedge p_2]\!]\, I\, \sigma = \mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge \mathcal{B}[\![p_2]\!]\, I\, \sigma \\
\mathcal{B}[\![p_1 \vee p_2]\!]\, I\, \sigma = \mathcal{B}[\![p_1]\!]\, I\, \sigma \vee \mathcal{B}[\![p_2]\!]\, I\, \sigma \quad \mathcal{B}[\![\neg p]\!]\, I\, \sigma = \neg \mathcal{B}[\![p]\!]\, I\, \sigma \\
\mathcal{B}[\![a_1\ rop\ a_2]\!]\, I\, \sigma = \mathcal{A}[\![a_1]\!]\, I\, \sigma\ rop\ \mathcal{A}[\![a_2]\!]\, I\, \sigma \quad \mathcal{B}[\![\forall i.p]\!]\, I\, \sigma = \forall n.(\mathcal{B}[\![p]\!]\, I[n/i]\, \sigma) \\
\mathcal{B}[\![\exists i.p]\!]\, I\, \sigma = \exists n.(\mathcal{B}[\![p]\!]\, I[n/i]\, \sigma) \quad \mathcal{B}[\![PC = L]\!]\, I\, \sigma = \sigma(PC) = L
\end{gathered}$$

Here, $I[n/i]$ stands for the interpretation that maps i to n but otherwise behaves like I.

We now give the semantic function $\mathcal{C}[\![\ ]\!]$ for assertions $p_1 \rightarrow p_2$. The definition uses the relation $\rightarrow^*$ on states, defined by $\sigma \rightarrow^* \sigma'$ if and only if σ' is reached from σ through zero or more state transitions (reflexive-transitive closure of the transition relation $\rightarrow$):

$$\mathcal{C}[\![p_1 \rightarrow p_2]\!] = \forall I, \sigma, \sigma'.[(\mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge \sigma \rightarrow^* \sigma') \implies \mathcal{B}[\![p_2]\!]\, I\, \sigma'] \tag{1}$$

Thus $p_1 \rightarrow p_2$ if, for each state σ where p_1 holds, and for each state σ' that is reachable from σ, p_2 holds for σ'.

▶ **Theorem 1** (Compositionality). $p_1 \rightarrow p_2 \wedge p_2 \rightarrow p_3 \implies p_1 \rightarrow p_3$.

Proof. Assume that $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_3$. $\sigma \rightarrow^* \sigma$ for all states σ. Thus, since $p_1 \rightarrow p_2$, p_2 holds for all states where p_1 holds. Since $p_2 \rightarrow p_3$ it follows that p_3 holds for each state reachable from a state where p_1 and thus also p_2 holds, thus it must hold that $p_1 \rightarrow p_3$. ◀

Theorem 1 implies that assertions for a program can be composed out of assertions on its parts, much like the inference rules for Floyd-Hoare logic.

7 Context Sensitivity

Value annotations and flow facts can be made more precise if context-sensitive. The contexts we have seen so far are sets of states defined by p' in an assertion $p \to (p' \implies p'')$, which loosens the requirement on p'' to hold merely for the reachable states where p' holds. However, another very important class of contexts are *calling contexts*. These can be used to express that a value annotation is to hold only when a function is called in a certain way, perhaps through a specific chain of other function calls. The well-known concept of *call-strings* [14] can be used to define such contexts.

In our setting a call-string is a sequence of special labels that identify particular program points like call sites for functions, or entry points to loops. Let $\mathbf{L}$ be the set of labels under consideration, and let $\mathbf{C} \subseteq \mathbf{L}$ be the set of labels that are considered to be call sites. A sequence $s \in \mathbf{C}$ is then a call-string. Let S be an expression that defines a set of call-strings $C(S) \subseteq \mathbf{C}$. The notation

$$p \to p' \textit{ through } S$$

is a suggested extension of the core language in Section 5 to denote an assertion where p' is to hold for those states, reachable from some state where p holds, through a sequence of states such that the sequence of traversed call sites belongs to $C(S)$. We will not be specific about the exact format of S: it could, for instance, be some kind of regular expression defining a set of call-strings.

For completeness we now give a formal semantics to this kind of assertion. In order to do this we need to develop some notation for different sequences. Given the transition relation "$\to$" on states, which describes the semantics of the host program, we define:

$$\begin{aligned}
\textit{Paths}(\sigma, \sigma') &= \{\, \sigma_1 \cdots \sigma_n \mid \sigma_1 = \sigma, \sigma_i \to \sigma_{i+1}, i = 1, \ldots, n-1, \sigma_n = \sigma' \,\} \\
PC(\sigma_1 \cdots \sigma_n) &= \sigma_1(PC) \cdots \sigma_n(PC) \\
\textit{Labels}(\sigma, \sigma') &= \{\, PC(\sigma_1 \cdots \sigma_n) \mid \sigma_1 \cdots \sigma_n \in \textit{Paths}(\sigma, \sigma') \,\}
\end{aligned}$$

$\textit{Paths}(\sigma, \sigma')$ is the set of sequences of states leading from σ to σ'. $PC(\sigma_1 \cdots \sigma_n)$ is the sequence of labels generated by the sequence of states $\sigma_1 \cdots \sigma_n$. $\textit{Labels}(\sigma, \sigma')$ is the set of sequences of labels generated by the possible state transitions leading from σ to σ'. Finally we introduce the well-known projection operator "$\upharpoonright$" on strings s and sub-alphabets A: $s \upharpoonright A$ is the substring of s obtained from its characters in A appearing in the same order as in s. For instance, if $A = \{a, b, c\}$ then $adecbbe \upharpoonright A = acbb$. We extend $\upharpoonright$ to sets of strings S, viz.

$$S \upharpoonright A = \{\, s \upharpoonright A \mid s \in S \,\}$$

We can now extend (1) to call-string-sensitive assertions as defined above:

$$\mathcal{C}[\![p_1 \to p_2 \textit{ through } S]\!] = \forall I, \sigma, \sigma'.[(\mathcal{B}[\![p_1]\!]\, I\, \sigma \wedge (\textit{Labels}(\sigma, \sigma') \upharpoonright \mathbf{C}) \cap C(S) \neq \emptyset) \implies \mathcal{B}[\![p_2]\!]\, I\, \sigma']$$

Thus the assertion $p_1 \to p_2$ *through* S holds if, for all sequences of labels from a state σ where p_1 holds to another state σ', such that the projection of the sequence onto the set of call sites is a call-string defined by S, p_2 holds for σ'. Compared with (1), where p_2 is to hold for all states reachable from a state satisfying p_1, p_2 now only has to hold for states reachable through a call-string given by S.

Theorem 1 can be extended to the context-sensitive case. Define $S \cdot S'$ by $C(S \cdot S') = \{\, s \cdot s' \mid s \in C(S), s' \in C(S') \,\}$, where "$\cdot$" is concatenation of sequences: we then have the following result (proof straightforward, but omitted due to lack of space):

▶ **Theorem 2.** $p_1 \to p_2 \textit{ through } S \wedge p_2 \to p_3 \textit{ through } S' \implies p_1 \to p_3 \textit{ through } S \cdot S'$.

8 Related Work

We have already reviewed the annotation languages of some WCET analysis tools in Section 2. There are a number of others: a comprehensive review and classification of such languages is found in [11]. Most of them express flow facts as linear arithmetic constraints on execution counters, as here, varying from simple looop bounds to general linear constraints. A notable exception is the Information Description Language [15], which can specify sets of feasible paths by regular expressions. This makes it possible to specify the exact order of execution of different program parts, whereas constraints in IPET execution counters only constrain the number of times they can execute and not the exact order.

Assertions using pre- and post-conditions have been used for a long time in formal software development: a classical example is the Vienna Development Method [2]. Such assertions can also be seen as *contracts*: the pre-condition is then the assumption on the environment, and the post-condition is what the program guarantees if the assumption is fulfilled. Contracts are essential for reasoning about component-based software. The language Eiffel provides means to express contracts [13].

On model level, the language OCL is used to specify properties of UML models. A formal semantics is given in [16]. The specification language Z uses Zermelo's set theory to express properties of models and programs in a pre/post-condition style [17].

9 Conclusions and Further Research

We have presented a simple core language for expressing assertions in pre/post-condition style, making minimal assumptions on the host language. This language can express value constraints, and it can be extended with IPET execution counters yielding the capability to express flow facts. Parametric flow facts and value constraints can be expressed using the auxiliary variables of the assertion language. We also proposed a way to include constraints with call strings, making is possible to express context-sensitive assertions. Special care was taken to develop the formal semantics of the language, and we proved a theorem about compositionality of assertions. This theorem is of practical interest since it allows assertions for a program to be composed from assertions on its parts.

We strongly believe that there is a need for a simple and general code-level assertion language that is designed from first principles, and has a clear semantics. In the best of worlds, a standardised such language could be used to exchange code-level information between a variety of tools including WCET analysis tools. In any case it can help understanding the underlying principles of annotation languages, and be used to give them a precise semantics.

Acknowledgment. This work was partially supported by COST Action IC1202: Timing Analysis On Code-Level (TACLe), and by the Swedish Research Council project Contesse (2010-4276).

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Proc. IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 35–46. Springer, October 2010.

2 Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: the Meta-Language*, number 61 in Lecture Notes in Comput. Sci. Springer-Verlag, 1978.

3 Armelle Bonenfant, Hugues Cassé, Marianne de Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX: A portable WCET annotation language. In *Proc. 20th International Conference on Real-Time and Network Systems (RTNS'12)*, pages 91–100, New York, NY, USA, 2012. ACM.

4 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 *Proc. IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 161–168, Kaohsiung, Taiwan, August 2008. IEEE Computer Society.

5 Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static memory and timing analysis of embedded systems code. In *3rd European Symposium on Verification and Validation of Software Systems (VVSS'07), Eindhoven, The Netherlands*, number 07-04 in TUE Computer Science Reports, pages 153–163, March 2007.

6 Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In Jan Gustafsson, editor, *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'03)*, Porto, July 2003.

7 R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proc. Symp. Applied Mathematics,* vol. 19: *Mathematical Aspects of Computer Science*, pages 19–32, Providence, R.I., 1967. American Mathematical Society.

8 C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

9 Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'02)*, 2002.

10 Raimund Kirner. The WCET analysis tool CalcWcet167. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 5th International Symposium on Leveraging Applications of Formal Methods (ISOLA'12)*, Lecture Notes in Comput. Sci., Heraclion, Crete, October 2012. Springer-Verlag.

11 Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software & Systems Modeling*, 10(3):411–437, 2011.

12 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. r-TuBound: Loop bounds for WCET analysis. In Nikolaj Bjørner and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Comput. Sci.*, pages 435–444. Springer, 2012.

13 Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

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

15 Chang Yun Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

16 Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *Lecture Notes in Comput. Sci.*, pages 449–464. Springer-Verlag, 1998.

17 J. Michael Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, 1988.

18 SWEET home page, 2011. `http://www.mrtc.mdh.se/projects/wcet/sweet/`.

19 Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction.* MIT Press, 1993.

A Formally Verified WCET Estimation Tool

André Maroneze[1], Sandrine Blazy[1], David Pichardie[2], and Isabelle Puaut[1]

1 **IRISA – Université Rennes 1**
Campus Universitaire de Beaulieu, Rennes, France
andre.maroneze@irisa.fr, sandrine.blazy@irisa.fr, isabelle.puaut@irisa.fr

2 **IRISA – ENS Rennes**
Campus Universitaire de Ker Lann, Bruz, France
david.pichardie@irisa.fr

Abstract

The application of formal methods in the development of safety-critical embedded software is recommended in order to provide strong guarantees about the absence of software errors. In this context, WCET estimation tools constitute an important element to be formally verified. We present a formally verified WCET estimation tool, integrated to the formally verified CompCert C compiler. Our tool comes with a machine-checked proof which ensures that its WCET estimates are safe. Our tool operates over C programs and is composed of two main parts, a loop bound estimation and an Implicit Path Enumeration Technique (IPET)-based WCET calculation method. We evaluated the precision of the WCET estimates on a reference benchmark and obtained results which are competitive with state-of-the-art WCET estimation techniques.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Formal Verification, CompCert C Compiler, WCET Estimation

Digital Object Identifier 10.4230/OASIcs.WCET.2014.11

1 Introduction

In the context of safety-critical embedded software, international regulations such as the DO-178C standard promote the use of formal methods for software development. Among them, formal verification provides guarantees about the specification and the implementation of a program through the use of machine-checked proofs. Instead of relying on a manual verification effort, the use of tools provides stronger guarantees about the absence of errors in the proof. This is especially important for reasoning on real languages such as C.

Safety-critical systems are an instance of real-time systems, where programs must respect given timing constraints. An important measure in real-time systems is the worst-case execution time of a program. Obtaining a *safe* WCET estimate (that is, a value at least as large as the actual WCET) is part of the necessary guarantees for such systems.

Current WCET estimation tools, even when based on sound static analysis techniques, are not verified. This may lead to bugs being accidentally introduced in the implementation. The main contribution of this paper is a formally verified WCET estimation tool operating over C code. It extends previous work on formally verified static analyses ([5] and [4]) by adding our WCET estimation, based on the classic IPET technique. In our approach, the code of our tool is automatically generated from its formal specification. Furthermore, machine-checked proofs ensure the estimated WCET is at least as large as the actual WCET.

Our formally verified WCET estimation tool has been integrated into CompCert [10], a moderately optimizing, formally verified C compiler usable for critical software. This

© André Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut;
licensed under Creative Commons License CC-BY

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 11–20

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

integration provides two major benefits for our tool: first, it allows us to reuse CompCert's formal specifications, including formal semantics for C and assembly (CompCert targets PowerPC, ARM and x86 assembly). Second, it enables the integration of analyses at different intermediate languages (e.g. high-level loop transformations and low-level WCET estimation). It also allows us to benefit from CompCert's optimizations (such as constant propagation) to improve the precision of our WCET estimates.

In our formal development, we verified two major components of our WCET estimation tool: a loop bound estimation technique (inspired by one of SWEET's [7] techniques for loop bound estimation), based on program slicing and a value analysis, and the generation of an integer linear programming (ILP) system for WCET estimation via IPET. Combining these techniques results in a WCET estimation tool together with a machine-checked proof that the produced WCET estimate is safe. We evaluated the precision and efficiency of our implementation on the Mälardalen WCET benchmarks [8]. Our results are competitive with state-of-the-art WCET estimation techniques.

This paper is structured as follows: in Section 2 we present the architecture of our WCET estimation tool, focusing on the techniques and implementation. In Section 3, we detail the proof architecture, presenting a high-level view of the final correctness theorem in our formalization and its main components. In Section 4, we describe the experimental evaluation and its results. We present some related work in Section 5 and then conclude in Section 6.

2 Architecture of our WCET estimation tool

We developed our WCET estimation tool within CompCert, which is equipped with a correctness theorem, that is, a proof relating the behaviors of source and compiled code, which ensures that no bugs are introduced during compilation. CompCert has several intermediate languages, among them RTL (for *Register Transfer Language*), where a program is represented by its control-flow graph (CFG). Our loop bound estimation is performed at this level. The WCET calculation phase is defined at the lowest level (assembly), closer to the executable for more precision. The result of our analyses performed in RTL is transported to assembly thanks to the correctness theorem.

Figure 1 illustrates the architecture of our WCET estimation tool (bottom row) within the CompCert compilation chain (top row). We start by presenting the loop bound estimation (Section 2.1). Due to space considerations, we only present an overview of this technique, referring the reader to [5] for more details.

The other main component in our tool is the WCET calculation step, which generates an ILP system to produce the WCET estimate (Section 2.2). It uses the result of the loop bound estimation. Finally, we present some (optional) loop transformations (Section 2.3), used to improve the precision of our WCET estimate. They are performed on structured loops, during front-end compilation.

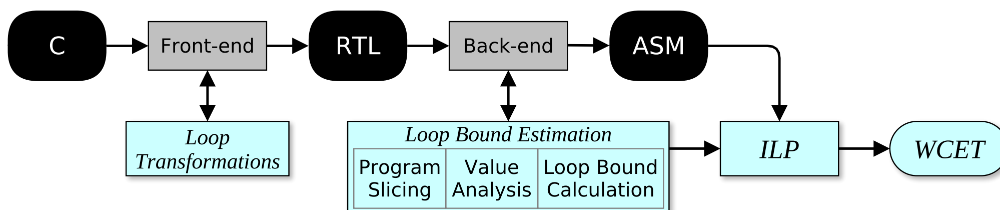


Figure 1 CompCert's compilation chain (top) and our WCET estimation tool (bottom).

2.1 Loop bound estimation

Our loop bound estimation technique, inspired by one of those used in SWEET [7], is composed of three parts: program slicing, value analysis, and loop bound calculation. In a deterministic and terminating program execution, the same program state cannot occur twice, i.e. the values of the program variables are unique at each iteration. Thus, we count the number of different states and use it as an upper bound for the number of loop iterations.

For each loop in the program, the first step consists in performing program slicing by removing statements not related to the loop header, which is our slicing criterion. This improves the precision and speed of the following passes. For instance, in the program below, where the loop header is the loop exit condition (`i < 5`), program slicing removes statements which do not affect the value of variable `i` (considering `f` free of side-effects).

```
i = 1;
a[0] = 0;
while (i < 5) {
    a[i] = a[i-1] + f(i);
    i++;
}
```

```
i = 1;

while (i < 5) {

    i++;
}
```

Then, our value analysis computes, for each program variable at each program point, an interval containing all possible values of that variable, as indicated in the code fragment below. This interval is a safe over-approximation, and therefore when counting the number of different possibilities, we will obtain a safe estimate of all possible values. This analysis is based on abstract interpretation, and it is detailed in [4]. An example program is presented below, with the result of our value analysis in the right column.

Code	Value analysis
`i = 0; j = 0;`	$i \in [0,0]$, $j \in [0,0]$
`while (i < 5)`	$i \in [0,5]$, $j \in [0,2]$
`{`	$i \in [0,4]$, $j \in [0,2]$
`    i++;`	$i \in [1,5]$, $j \in [0,2]$
`    if (i == 5 && j < 2) {`	$i \in [5,5]$, $j \in [0,1]$
`      i = 0;`	$i \in [0,0]$, $j \in [0,1]$
`      j++;`	$i \in [0,0]$, $j \in [1,2]$
`    }`	$i \in [0,5]$, $j \in [0,2]$
`}`	$i \in [5,5]$, $j \in [2,2]$

The final stage consists in computing the product of the size of the domains of the *relevant variables* for each loop – variables which are *live*, *used* and *modified* inside the loop. Only these variables may influence the number of loop iterations.

The computed product is a safe loop bound estimation. For instance, in the previous example there are two relevant variables, `i` and `j`. The number of iterations is safely bounded by the product of the sizes of their intervals at the loop exit condition ($[0,5]$ and $[0,2]$), that is, $6 \times 3 = 18$ iterations (the exact bound here is 15 iterations). This method also works for nested loops, by computing the product of outer and inner loop bounds.

2.2 IPET-based WCET estimation

We apply a classic technique, namely IPET [11], to produce an ILP system representing the program's execution flow. The objective function to maximize, representing the program execution time, is $T = \sum_{i \in code} t_i x_i$. For each program point i in the program, x_i is a static over-approximation of the number of times i is executed, and t_i is the cost coefficient (in cycles) associated to this program point. In our tool, we currently use a simple cost model

where $t_i = 1$ for every instruction. Our ILP constraints are obtained from the reconstructed CFG at the assembly level. The ILP also incorporates the loop bounds previously computed, as constraints of the form $x_h \leq N$, where h is a loop header and N is the inferred loop bound. RTL bounds are correctly transported to assembly thanks to CompCert's semantic preservation theorem.

2.3 Loop transformations

To improve the precision of our loop bound estimations, we apply two kinds of loop transformations: *loop inversion* and *loop unrolling*. Loop inversion consists in converting `while` and `for` loops into `do-while` loops. The motivation behind this transformation is the fact that each kind of loop behaves differently with respect to loop iterations (`while` and `for` loops execute the loop exit condition more often than the loop body). A cost-effective way to deal with the variety of C constructs is to reduce them to a few general cases and treat them uniformly, which is done by loop inversion. The code fragment below illustrates the application of loop inversion to a simple `while` loop.

```
while (i < 5) {
    f(i);
    i++;
}
```

```
if (i < 5) {
    do {
        f(i);
        i++;
    } while (i < 5);
}
```

Loop unrolling is used to improve the precision of the WCET estimate for loops containing conditional branches with different execution costs, for instance in a loop where the first iteration performs differently from the others. Without loop unrolling, the cost of the longest branch is considered in each iteration and results in a WCET overestimation. The code below illustrates an example where loop unrolling helps to improve precision. The values inside the `/*...*/` comments indicate statically known values which will be optimized by a constant propagation pass. For instance, the second call to the `init()` function in the unrolled loop below is unreachable, and therefore eliminated after code simplification, as indicated in the last column.

```
i = 0;
do {
    if (i == 0) init();
    i++;
} while (i < 2);
```

```
i = 0;
do {
    if (i /*0*/ == 0) init();
    i++;
    if (i /*1*/ >= 2) break;
    if (i /*1*/ == 0) init();
    i++;
} while (i /*2*/ < 2);
```

```
i = 0;
do {
    init();
    i++;
    //if (1 >= 2) break;
    //if (1 == 0) init();
    i++;
} while (i < 2);
```

To avoid excessive unrolling, we only unroll loops with conditional branches (which can help improve the WCET), and limit the unrolling factor according to the size of the code.

3 Proof of our WCET estimation tool

Our tool has been specified and formally proved correct using the Coq [6] proof assistant. With Coq's functional specification and programming language, we proceeded as follows: first, we specified our functions; then, we defined logical properties about these specifications; afterwards, we proved these properties using Coq's interactive proof mechanism, where we

write the proof step-by-step while Coq checks its correctness; finally, we used the automatic code generation mechanism available in Coq to obtain our verified tool directly from its specification. In the end, we obtained an executable software (our WCET estimation tool) plus its *proof of correctness*.

The proof of correctness is a proof of semantic preservation. In this section, we define more precisely our notion of semantic preservation and then we detail the proof architecture of the main components of the tool. We present *what* has been proved correct, with an intuitive notion of the main correctness lemmas, and briefly mention the proof techniques, i.e. *how* it has been proved.

3.1 Correctness theorem of the WCET estimation

To define the correctness of a WCET estimation algorithm, we need to define the WCET itself and then what is a WCET estimate and how to compute it. We do so from a formal semantics of the CompCert assembly language. We present here some notions necessary for understanding our proof sketches.

In our semantics, a *program state* contains the value of each memory variable and machine register at a given point in the program execution. The semantics defines the evolution of program states. A well-formed sequence of program states forms an *execution trace*. We denote $\mathit{Terminates}(P, \mathit{tr})$ as the complete execution of program P, producing the execution trace tr.

We extended the CompCert assembly semantics to take into account the quantitative aspect of the WCET, adding *execution counters* for every program point and program transition (CFG vertex and edge, respectively). These counters correspond to the number of occurrences of the program point (or program transition) in a given execution trace. They represent the *exact* values obtained during execution along that trace.

The execution time T of an execution trace tr is defined as *the sum of the execution counters of every program point* (since local costs are considered as 1). The *worst-case* execution time of a program P is thus defined as the maximum execution time among all possible program executions. More formally, we can define the WCET as follows. Let $\mathit{TR}(P)$ be the set of all possible traces of program P, that is, $\mathit{TR}(P) = \{\mathit{tr} \mid \mathit{Terminates}(P, \mathit{tr})\}$. Then:

$$\mathit{WCET}(P) = \max_{\mathit{tr} \in \mathit{TR}(P)} T(\mathit{tr})$$

Note that this is only a mathematical definition: neither our algorithm nor our proof actually enumerates all program paths.

To perform the WCET estimation, we *over-approximate* the execution counters using the x_i variables of the ILP system. Our WCET estimate, WCET_E, is the sum of all x_i variables. For a WCET estimation tool to be considered *sound*, all estimates it produces must be larger than or equal to the actual WCET. This can be stated as follows: *for any terminating program, every estimate t_E produced by the tool must be an over-approximation of the actual WCET.*

▶ **Theorem 1** (Correctness of the WCET Estimation)**.**

$$\forall P, \forall \mathit{tr}, \mathit{Terminates}(P, \mathit{tr}) \wedge \mathit{WCET}_E(P) = \lfloor t_E \rfloor \Longrightarrow \mathit{WCET}(P) \leq t_E$$

WCET_E is the actual WCET estimation (partial) function, defined as the composition of all stages of our WCET estimation tool. A successful estimation is denoted by $\lfloor t_E \rfloor$. The executable code for WCET_E is automatically obtained from its formal specification.

3.2 Proof techniques

In formal verification, the standard approach consists in formally specifying and proving every concept and algorithm, once and for all. For instance, this means formally verifying an ILP solver to prove the correctness of the WCET calculation phase, for every program. However, the formal verification of an ILP solver is an endeavor which is out of our approach. In such situations, there is an alternative proof technique which provides strong guarantees about correctness, called *a posteriori validation* [14]. It consists in checking the *result* of a computation (using a *validator*) without proving each step of its construction.

More specifically, we used *verified validation*, which includes a proof of correctness of the validator itself. This proof ensures that any result accepted by the validator is indeed correct. This technique is already used in CompCert, for instance during register allocation.

The major advantages of a posteriori validation are (1) manageable proof effort (especially for algorithms relying on sophisticated heuristics, whose proof might be otherwise too costly) and (2) the possibility to integrate untrusted code (such as an off-the-shelf ILP solver, instead of having to prove the solver itself). The trade-off is that validation may incur some extra computation time during program execution. However, in our case validators were used in situations where their cost was negligible with respect to the computation time of the solution itself (such as during ILP computation).

3.3 Correctness proof of the loop bound estimation

To deal with all kinds of loops (such as unstructured loops created by `goto` statements), the correctness theorem of our loop bound estimation is defined in terms of arbitrary program points. In other words, we prove that the execution counters of a given set of program points are bounded by our estimation technique. In practical terms, these program points correspond to *loop headers*, which entails that their loops are effectively bounded.

Since the loop bound estimation is composed of three parts, we defined a correctness theorem for each of them and combined the proofs using theorems that already exist in CompCert. The idea behind the correctness of each of these intermediate theorems is presented in the following.

Program slicing

Informally, the correctness of program slicing is stated as: *the bounds computed in the sliced program are a safe overestimation of the bounds computed in the original program.* In other words, we can transform a program P, obtaining a program slice P', compute bounds on the latter, and safely transpose these bounds to the original program.

More formally, and as an example of a Coq theorem, we present below a simplified version of the correctness theorem of program slicing. Let `P` be an RTL program and `slice(P,i)` its sliced version with respect to program point `i`. Let `tr` and `tr'` be valid execution traces of `P` and `slice(P,i)`, respectively. Then, any bound `B` which is correct for the counter of `i` in `P'` (that is, `counter(tr', i) ≤ B`) is also a correct bound in the original program.

```
Theorem slicing_correctness:
   forall (P : program) (i : program_point) (tr tr' : exec_trace),
     Terminates(P, tr) ∧ Terminates(slice(P,i), tr') ⟹
   forall (B : int),
     counter(tr', i) ≤ B
     ⟹ counter(tr, i) ≤ B.
```

Program slicing is a transformation that preserves the semantics with respect to some slicing criterion. This property is classically proved by induction on the execution relation between the original and transformed programs. It amounts to showing that a step-by-step parallel execution of both programs results in the same execution counters for the slicing criterion (i.e. the loop header) in both programs.

To compute a slice efficiently, sophisticated data structures (such as postdominator trees and program dependence graphs) are necessary. To avoid formalizing all of them, we developed an untrusted program slicer and validated its result, obtaining the same guarantees while enabling the adoption of more efficient slicers without having to change the proof. Detailed information about the program slicer and its validation are described in [5].

Value analysis

An intuitive notion of the formal correctness of the value analysis can be stated as follows: *for any interval given by the value analysis, all possible variable values are taken into account.* This is true for each program point and each program variable. To prove it correct, we first show that each individual execution step (given by the assembly formal semantics) is correct in itself. For instance, after a CFG branch merge, an interval union of the values of each branch is a correct over-approximation of the values after the merge.

The major difficulties in proving the value analysis come from issues related to the complexity of the C language (such as having a large number of operators) and the efficiency of the analysis. In the presence of loops, the analysis needs to perform several iterations before it reaches a final (stable) solution. We show that each operator is correctly abstracted into an interval, and then we show that the final solution is stable with respect to loops. Both facts entail that the solution is a correct over-approximation.

Loop bound calculation

The proof of the loop bound calculation relates the sizes of the intervals of variable values to the execution counters in the extended semantics. It is proved by induction on the execution trace: every time an execution step would allow a program point to exceed its bounds, this would lead to an infinite loop. The bounds, defined with respect to the result of the value analysis, contain every possible value for all variables which influence the loop condition. For instance, if variables i and j have bounds $[0, 1]$, and they are the only variables influencing the loop exit condition, then these pairs can appear at most once in the execution trace: $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. Reaching the loop exit condition a fifth time (e.g. with $(i, j) = (1, 1)$) would imply an infinite repetition of the same sub trace, leading to an unbounded execution time. The same reasoning is extended to handle nested loops.

3.4 Correctness proof of the ILP

As mentioned in Section 3.2, we use *a posteriori* validation to guarantee a correct ILP result. Our verification is decomposed in three parts, as indicated in Figure 2. The first part consists in proving that the ILP generation is correct. The second part is the ILP solving, performed by an external component. The final part is the validation of the solution.

Proving the ILP correct means showing that each ILP variable is an over-approximation of its corresponding execution counter. Using the flow constraints (including loop bounds), the proof of correctness is based on reasoning by induction on the execution trace.

The generated ILP is sent to an external ILP solver (such as lpsolve [2]), which returns an assignment of variable values and the corresponding WCET estimate. We prove two

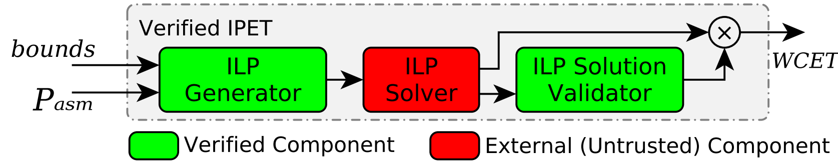


Figure 2 Diagram of the components used for the IPET verification.

properties about the result: that (1) it is a solution of the system (a valid assignment), and that (2) it is the *largest* solution (i.e. the worst-case).

Verifying property (1) only requires substituting variables with their assigned values and checking that all constraints are respected. To verify property (2), we show that any larger solution is infeasible. To do so, we augment the system with the negation of the solution (i.e. we add $\sum x_i > t_E$, where t_E is the solution given by the solver) and then we compute a *Farkas certificate* [3], a set of linear coefficients which can be used to prove the infeasibility of a linear system. This computation amounts to the solution of a system of the same size.

To integrate this technique within the proved framework, we define and prove a verified validator in Coq, whose inputs are the ILP system, its (untrusted) solution and a certificate, and whose output is *true* if the certificate confirms that the solution is valid. The correctness theorem of the validator ensures that, if the inputs pass validation, then the solution is a correct WCET estimate. With this final step, we prove the theorem stated in Section 3.1.

3.5 Feedback on the proof effort

Our proof effort resulted in over 15,000 lines of Coq code (half of them being Coq definitions, and the other half being Coq proofs). The development also contains about 2,000 lines of manually written OCaml code (which includes code such as the program slicer), and about the same size of code automatically generated from the Coq development.

Concerning the development methodology, we followed the standard practice in formal verification, which consists in performing the specification and the proof in parallel. While performing the proof, several details about the specification need to be reformulated, either to improve their clarity, or to enable the proof to go through (e.g. there are several ways to specify a program slicer, but only a few of them lead to efficient proof strategies).

4 Experimental evaluation

We evaluated our WCET estimation tool on the Mälardalen WCET benchmarks. We considered 15 of the 20 programs evaluated in [5]. The other 5 programs (`adpcm`, `fft1`, `fir`, `insertsort` and `ludcmp`) contain loops which were not bounded due to imprecisions in the value analysis, such as loops depending on floating-point variables or on memory contents.

We compiled the code to PowerPC assembly to estimate its WCET. To evaluate the precision of our WCET estimation, we modified the programs having several possible execution paths (e.g. by setting specific values to input variables) to ensure execution of a worst-case path. We executed them using the formal semantics to obtain the exact WCET, and then we compared this value to the estimate obtained on the original program.

Figure 3 presents our evaluation. For each program, we indicate the size of its source code (LoC) and we present the relative WCET overestimation, using 3 different configurations: no loop transformations, loop inversion only, and loop inversion together with loop unrolling. We also present the analysis times of our tool.

		No Loop Transformations		Loop Inversion		Inversion+Unrolling		
Program	***LoC***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Class***
cnt	267	18.3%	0.1	2.8%	0.2	3.3%	7.0	OK
cover	640	10.9%	1.0	11.5%	1.0	0.0%	21.8	OK
crc	128	100.2%	0.2	99.5%	0.2	99.2%	1.7	Imprecise
edn	285	141.5%	12.5	110.4%	13.1	110.4%	23.4	Imprecise
expint	157	2601.6%	0.0	2419.7%	0.0	0.0%	8.2	OK
fdct	239	0.0%	0.4	0.0%	0.5	0.0%	0.6	OK
fibcall	72	0.9%	0.0	1.1%	0.0	1.1%	0.0	OK
jfdctint	375	0.0%	0.3	0.0%	0.3	0.0%	0.5	OK
lcdnum	64	50.9%	0.0	55.2%	0.0	11.9%	0.1	OK
matmult	163	11.5%	0.3	0.0%	0.3	0.0%	0.5	OK
ndes	231	12.2%	4.0	3.6%	4.2	3.6%	225.4	OK
ns	535	88.3%	0.1	0.2%	0.1	0.2%	0.2	OK
nsichneu	4,253	106.1%	60.5	106.1%	60.2	106.3%	89.7	Imprecise
qurt	166	168.2%	0.7	165.7%	0.7	215.2%	3.0	Imprecise
ud	161	225.1%	0.6	217.3%	0.6	265.2%	11.3	Imprecise

Figure 3 Experimental results of our WCET tool, given as a relative overestimation w.r.t. the exact WCET, without and then with loop transformations.

We classify the programs in two groups: OK (WCET estimate with no or small overestimation) and Imprecise (significant overestimation). Comparing the different configurations confirms that the loop transformations improved the precision of the WCET estimate, sometimes drastically (e.g. `ns` goes from 88% overestimation down to 0%, thanks to loop inversion, and `expint` goes from 2420% to 0% due to loop unrolling). In a few programs, we see a *relative* increase, which is due to the decrease in the *absolute* WCET of the transformed loops. Overall, the loop optimizations provide a significant benefit in terms of precision.

5 Related work

There are several WCET estimation tools in the literature which perform loop bound estimation, such as aiT [9], Bound-T [16], oRange [13], SWEET [7] and TuBound [15]. Our objective is not to develop a novel technique to estimate the WCET, but to formally specify an existing method and to prove it correct.

Due to its extensive flow analysis, SWEET was the inspiration for our loop bound estimation. SWEET has two loop bound estimation techniques: one similar to the one we formally verified, and another one having a context-sensitive mechanism capable of inferring more precise flow constraints. The trade-off is that, in some cases, it may perform excessive unrolling and not terminate. Unlike our tool, SWEET is not formally verified.

WCC [12] is a C compiler integrating an external WCET estimation tool. WCC performs automatic loop bound estimation based on polyhedral evaluation, but it focuses on hardware-level optimizations, while we focus on the control-flow analysis. WCC is not formally verified and it relies on other tools (such as aiT) to obtain WCET estimates. Our tool, on the other hand, only relies on external tools if they can provide certificates. This ensures that bugs in their implementation do not affect the correctness of our tool.

The CerCo [1] project shares our views about the necessity of formal guarantees in WCET estimation, but it has a different approach and objective. In CerCo, an original technique transports annotations from the assembly to the C source program, and then it relies on a non-verified tool, based on program proof (Frama-C), to produce proof certificates about the correctness of the WCET estimates. Unlike ours, this approach is not entirely automatic: after analyzing a program, some verification conditions may need manual proof. Concerning the hardware model, cost information is based on a simple timing model, like our tool.

6 Conclusion

We presented a formally verified WCET estimation tool whose correctness theorem ensures its WCET estimates are safe. Our tool is built within the CompCert C compiler, providing extra guarantees about the execution time of the code produced by the compiler. There are now two complementary tools which operate over C code for safety-critical embedded systems, and these tools are formally verified, which provides unprecedented guarantees.

Our tool relies on the formal verification of different techniques commonly used by industrial-strength WCET estimation tools: loop bound estimation and IPET. Experimental evaluation of the precision of our tool indicates satisfactory results. Future work includes improving the precision of current analyses and adding a more realistic hardware model.

References

1. R. Amadio, A. Asperti, N. Ayache, B. Campbell, D. P. Mulligan, R. Pollack, Y. Régis-Gianas, C. S. Coen, and I. Stark. Certified complexity. *Procedia CS*, 7:175–177, 2011.
2. M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve : Open source (Mixed-Integer) Linear Programming system. `http://lpsolve.sourceforge.net`.
3. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *TGC*, pages 253–267. Springer, 2010.
4. S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, LNCS, pages 324–344. Springer, 2013.
5. S. Blazy, A. Maroneze, and D. Pichardie. Formal verification of loop bound estimation for WCET analysis. In *VSTTE 2013*, LNCS. Springer, 2013.
6. Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1989–2014.
7. A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.
8. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 137–147, 2010.
9. R. Heckmann and C. Ferdinand. aiT: worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions*, pages 377–384, 2004.
10. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
11. Y. T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CADICS*, 16(12):1477–1487, 1997.
12. P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011.
13. M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proc. of ERTSS*, pages 161–166. IEEE Computer Society, 2008.
14. G. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.
15. A. Prantl, M. Schordan, and J. Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *WCET*, 2008.
16. Tidorum. Bound-T tool homepage. `http://www.bound-t.com`, 2010.

On the Sustainability of the Extreme Value Theory for WCET Estimation

Luca Santinelli[1], Jérôme Morio[1], Guillaume Dufour[1], and Damien Jacquemart[2]

1 ONERA – The France Aerospace Lab, Toulouse, name.surname@onera.fr
2 ONERA – The France Aerospace Lab, Palaiseau and INRIA Rennes ASPI

Abstract

Measurement-based approaches with extreme value worst-case estimations are beginning to be proficiently considered for timing analyses. In this paper, we intend to make more formal extreme value theory applicability to safe worst-case execution time estimations. We outline complexities and challenges behind extreme value theory assumptions and parameter tuning. Including the knowledge requirements, we are able to conclude about safety of the probabilistic worst-case execution estimations from the extreme value theory, and execution time measurements.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems, C.4 Performance of Systems, G.3 Probability and Statistics

Keywords and phrases Extreme Value Theory, Worst-Case Execution Time, Probabilistic Worst-Case Execution Time, Dependence, Stationarity

Digital Object Identifier 10.4230/OASIcs.WCET.2014.21

1 Introduction

The measurement-based probabilistic timing analysis composes of measurements and statistic analyses, particularly the Extreme Value Theory (EVT), to outline rare events out of execution time measurements.

The EVT is a branch of statistics dealing with the extreme deviations from the median of probability distributions. It seeks to assess, from an ordered sample of a given random variable, the probability of events that are more extreme than any previously observed. The EVT applies to execution time and Worst-Case Execution Time (WCET), since it offers guarantees to have projected the tail of measurement distributions. Thus, EVT provides what is called safe WCET estimations accounting for rare events.

Real-time relies on WCETs to model task execution behaviors: a real-time system becomes predictable by always accounting for the worst-case at every task execution. As the input space for the task code is finite and the hardware behavior is assumed to be deterministic, it is reasonable to argue about the exact worst-case execution time. The WCET_{exact} and its estimation $\overline{C}$ are upper-bounds for any possible execution behavior of that task.

Unfortunately, systems are unpredictable as the environment can be diverse and dynamic with multiple possible evolutions in time. Both hardware and software elements may experience some variability or even randomness[1], e.g. multi-processor, cache, branch predictors, DRAM refresh, interruptions that occur whenever they are most inappropriate, and the interferences between interacting elements in the system lead to the dependences

[1] Randomness is intended in the common sense, as lack of pattern or predictability in events.

© Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 21–30

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

that emphasize unpredictabilities and variability, [7]. WCET_{exact} is in general unknown and potentially unknowable, and the estimation $\overline{C}$ could be extremely pessimistic.

Probabilities could model system indeterminacy and unpredictabilities, capturing multiple behaviors with their frequencies of happening. Such a fine grained system probabilistic representation reduces the pessimism brought by deterministic models, where only the estimation of the worst-case is taken into account. Then, the challenge is to guarantee probabilistic estimations of worst-case execution time, and build with the probabilities a safe alternative to the deterministic real-time. The EVT has been recently applied with that purpose, but it is still far from accomplishing such job.

1.1 State of the Art

A measurement-based approach to timing analysis implies a near-zero cost for system coding and modeling in order to get task execution time observations [2]. The problem consists of guaranteeing the coverage of all the possible execution conditions and obtaining reliable WCET estimations.

Ensuring exhaustive execution condition coverage requires knowledge of the system, slightly reducing the advantage of measurement-based approaches with respect to static timing analysis approaches [14]. Even if the worst-case execution time is computed using test generation techniques which ensure feasible path coverage, usual assumptions reduce the input set variability. This fact decreases complexity, but demands an improved system model to identify the worst-case execution condition.

More "analytical" approaches to measurement-based timing analysis, make use of the statistics of extremes [8, 9] to construct predicted WCETs. In particular, recent works have re-formalized the application of EVT to the WCET problem [2, 1] by making use of an ad-hoc probabilistic hardware architecture. Those approaches are able to guarantee accurate probabilistic WCET estimations from measured execution time distributions.

Contributions. In this article, we intend to show challenges and possibilities applying the EVT to the task execution problem. Thus, we present the EVT resulting distribution for different system conditions and parameters. Besides, we aim at continuing the discussion around claimed robustness of the extreme value theory in terms of the guarantees it offers to the probabilistic worst-case execution time estimation.

The idea of this work is to present some results from a set of experiments to show: i) the impact of EVT parameters on the resulting worst-case execution time distributions; ii) the differences between block maxima EVT and peak over threshold EVT, the two approaches to EVT; iii) a qualitative evaluation of the EVT robustness. We focus on the required EVT hypotheses and their impact on the resulting pWCET estimations. The whole statistical analysis framework is to begin a complete and formal discussion about EVT sustainability to the execution time problem.

Data Setup. For our tests we make use of real traces taken from an Intel(R) Xeon(R) E5620 2.4 GHz dual socket, each socket with four cores and three levels of cache. The schedMcore[2] runtime support and Linux Trace Toolkit new generation (LTTNG) tracing framework are applied to guarantee real-time execution and extract accurate execution time measurements. The task implementations considered are single-path and multi-path tasks from the Mälardelen benchmark suite[3]. Out of them we extract the execution time measurements as traces called "single-path" and "multi-path", respectively.

[2] `https://forge.onera.fr/projects/schedmcore`
[3] `http://www.mrtc.mdh.se/projects/wcet`

Furthermore, we apply two artificial execution time random distributions as the normal distribution and a multi-modal distribution (obtained from 3 normal random variables combined). They are considered to compare results with realistic measurements. All the execution time are in *nsec*, and the number of observations is 100000 per trace.

2 Probabilistic Timing Analysis

Execution Time Profiles (ETPs) $\mathcal{C}$ are measured execution time distributions.[4] $\mathcal{C}$ are empirical distributions that lack of completion and coverage, which means that from the ETPs it is not possible to conclude about worst-case execution time. Nonetheless, ETPs are important to investigate in order to derive execution pattern/trends from which define worst-case execution conditions, and then worst-case execution times.

2.1 Probabilistic Worst-Case Execution Time

Statistical estimations of the worst-case execution time induce the notion of probabilistic WCET (pWCET), alternative to the deterministic WCET. pWCET is as distributions of WCET values C_j, each of them with associated a probability p_j of being WCET, $\mathcal{C} = \begin{pmatrix} C_j \\ p_j = P\{\mathcal{C} = C_j\} \end{pmatrix}_{j \in \{1,\cdots,J\}}$; p_j is the probability that C_j is an upper-bounds of the task execution time. The following is a possible definition of pWCET.

▶ **Definition 1** (probabilistic Worst-Case Execution Time). Given $\mathcal{C}_k$ the distribution of execution time measured in a certain configuration/condition k. The probabilistic Worst-Case Execution Time distribution $\mathcal{C}^*$ of a task is a tight upper bound on the execution time $\mathcal{C}_k$ of all possible execution conditions. Hence, $\forall k,\ \mathcal{C}^* \succeq \mathcal{C}_k$.[5]

The exact pWCET $\overline{\mathcal{C}}^*$ would be the tightest upper bound to any $\mathcal{C}_k$, $\mathcal{C}^* \succeq \overline{\mathcal{C}}^* \succeq \mathcal{C}_k,\ \forall k$.

The probabilistic worst-case execution time can also be defined from the exceeding thresholds and the 1-Cumulative Distribution Function (1-CDF) representation. Given a probability of exceedence p^*, the value C^* is the worst-case execution time such that $P\{\mathcal{C}^* \geq C^*\} \leq p^*$. Alternative to the pWCET distribution, we can call minimum probabilistic worst-case execution time the tuple $\langle C^*, p^* \rangle$. In real-time and certification issues, safety is validated with exceedence probability smaller than 10^{-9}. We consider that threshold and define the pWCET as $\langle C^*, 10^{-9} \rangle$, although the following reasoning is open to any threshold.

A pWCET estimation $\mathcal{C}^*$, in order to be safe, has to be greater than or equal to the exact pWCET, which is unknown, and any measurement $\mathcal{C}_k$. A distribution $\mathcal{C}^*$ is greater than or equal to a distribution $\mathcal{C}_k$ iff $P\{\mathcal{C}^* \leq c\} \leq P\{\mathcal{C}_k \leq c\}$ for any c and the two random variables are not identically distributed (two different distributions). For the $\langle C^*, 10^{-9} \rangle$ estimation, we say that it is safe if for each pWCET $\langle C, p \rangle$, including the exact one, for all $C \geq C^*$, $p \leq 10^{-9}$.

2.2 Extreme Value Theory

The extreme value theory is a branch of statistics dealing with the extreme deviations from the median of probability distributions. It seeks assessing, from a given ordered sample of

[4] We use calligraphic letters to represent probability distributions; non calligraphic letters are for single values.

[5] Total ordering of distributions is guaranteed by comparing distribution probabilities. Thus a distribution $\mathcal{C}_i$ is greater than or equal to a distribution $\mathcal{C}_j$, $\mathcal{C}_i \succeq \mathcal{C}_j$, iff $P\{\mathcal{C}_i \leq c\} \leq P\{\mathcal{C}_j \leq c\}$, for every c.

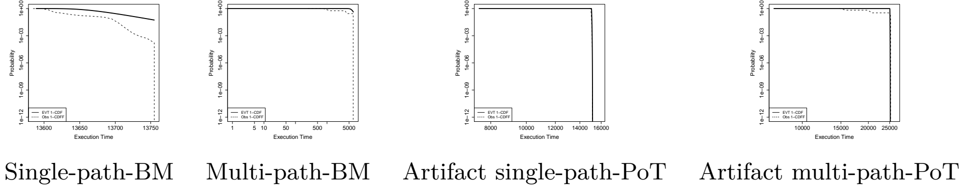


Figure 1 EVT applied to different cases.

a random variable, the probability of events that are more extreme than any previously observed. The EVT allows to estimate tails of distributions and thus explores rare events, where the WCET and its probabilistic version pWCET should lie. Two are the EVT approaches possible: Block Maxima (BM) and Peak over Threshold (PoT). The safety of the statistical pWCET estimation through EVT has always been referred to the independence and identical distribution (i.i.d.) hypotheses. If both are verified, then the EVT distribution tail projection can be considered as a safe pWCET estimation. Along this paper we discuss those assumptions and extend them.

Figure 1 shows the tail projection effect of the EVT once applied to measured distributions. With the 1-CDF representation we appreciate not strong difference between the measurements distributions and the EVT distributions. For the scope of the paper, this is what we call accuracy of the EVT estimations.

2.2.1 Law of sample maxima – Block maxima approach

EVT is notably very useful when one has to work with only a fixed set of data (measurements/observations), not having other info outside those observed. Consequently it is assumed in the following that a set of i.i.d. samples $X_1, \ldots, X_N$ of a time series $(X_t)_{t>0}$ (equivalently as distribution $\mathcal{X}$ being the samples from distributions[6]) is available. The associated ordered sample set is defined with $X_{(1)}, \ldots, X_{(N)}$. EVT enables to estimate for some thresholds S the probability $P\{(X_t)_{t>0} > S\}$.

The main result of EVT [5], is that the maxima of an i.i.d. sequence converges to a Generalized Extreme Value (GEV) distribution G_ξ under some general conditions, which admits a generic Cumulative Distribution Function (CDF) $\mathcal{G}_\xi(x)$. GEV distributions are composed of three distinct types, characterized by $\xi = 0$, $\xi > 0$ and $\xi < 0$ that correspond to the Gumbel, Fréchet and Weibull distributions respectively. Let us define $\mathcal{G}$, the CDF of the i.i.d. samples $X_{(1)}, \ldots, X_{(N)}$.

▶ **Theorem 2.** *Suppose there exist a_N and b_N, with $a_N > 0$ such that, for all $y \in \mathbb{R}$ $P\left\{\frac{\mathcal{X}_{(N)}-b_N}{a_N} \leq y\right\} = \mathcal{G}^N(a_N y + b_N) \overset{N\to\infty}{\longrightarrow} \mathcal{G}(y)$, where $\mathcal{G}$ is a non degenerate CDF, then $\mathcal{G}$ is a GEV distribution $\mathcal{G}_\xi$. In this case, one denotes $\mathcal{G} \in MDA(\xi)$ (MDA = Maximum Domain of Attraction).*

Unless samples of maxima are directly available, it is then required to group the samples $X_{(1)}, \ldots, X_{(N)}$ into blocks and fit the GEV using the maximum of each block. While the a_N and b_N distribution parameters are found by best fitting the input trace of events, the grouping into block maximum is somewhat an arbitrary parameter. Although all three

[6] A sample, equivalently an observation, comes from a distribution, thus the representation as calligraphic or non-calligraphic letters are equivalent.

parameters are influent, we focus on the block size, named b from now on, to show the impact that it has on the EVT pWCET estimation.

2.2.2 Peak over threshold approach

Instead of grouping the samples into block maxima, PoT considers the largest samples X_i to estimate the probability $P\{\mathcal{X} > S\}$. The link between EVT and the distribution of a threshold exceedence is firstly described in [13]. The following theorem can be obtained:

▶ **Theorem 3.** *Let us assume that the distribution function $\mathcal{G}$ of i.i.d. samples $X_1, \ldots, X_N$ is continuous. Set $y^* = \sup\{y, \mathcal{G}(y) < 1\} = \inf\{y, \mathcal{G}(y) = 1\}$. Then, the next two assertions are equivalent: a) $\mathcal{G} \in MDA(\xi)$, and b) there exists a positive and measurable function $u \mapsto \beta(u)$ such that $\lim_{u \mapsto y^*} \sup_{0<y<y^*-u} |\mathcal{G}^u(y) - \mathcal{H}_{\xi,\beta(u)}(y)| = 0$. $\mathcal{G}^u(y) = P\{\mathcal{X} - u \leq y | \mathcal{X} > u\}$, and $\mathcal{H}_{\xi,\beta(u)}$ is the CDF of a generalized Pareto distribution (GPD) with shape parameter ξ and scale parameter $\beta(u)$.*

The expression of the GPD distribution function $\mathcal{H}_{\xi,\beta}(x)$ depends on both ξ and β.

Theorem 3 is in fact useful to estimate a probability of exceedance $P(\mathcal{X} > S)$ since it can be rewritten as $P\{\mathcal{X} > S\} = P\{\mathcal{X} > S | \mathcal{X} > u\} \cdot P\{\mathcal{X} > u\}$, for $S > u$. A natural estimate of $P\{\mathcal{X} > u\}$ is given by the empirical mean $\hat{P}^{MC}\{\mathcal{X} > u\} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{X_i > u}$, from the Monte Carlo method (MC). With the Theorem 3 and for significant value of u, one obtains $\hat{P}\{\mathcal{X} > S | \mathcal{X} > u\} = 1 - \mathcal{H}_{\xi,\beta(u)}(S - u)$. The estimate of $P\{\mathcal{X} > S\}$ is then built with $\hat{P}^{POT}\{\mathcal{X} > S\} = \left(\frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{X_i > u}\right) \cdot \left(1 - \mathcal{H}_{\xi,\beta(u)}(S - u)\right)$.

The parameter u can be selected arbitrarily, as it affects the accuracy as well as safety of the EVT PoT estimation.

3 Extreme Value Theory applicability

First we describe what is the independence we are looking for in order to apply the EVT, then we extend it to other conditions. The meaning of independence we are looking for is whether individual observations within the same execution trace are correlated with each other or not. Knowing one observation tells you something about another, in which case they are dependent; knowing one observation tells you nothing about another, in which case they are independent. In this section we investigate deeply stationarity and independences/dependences for execution time traces.

The independence is not a necessary hypothesis for the EVT, since Leadbetter et al. [11] and Hsing [10] developed EVT for stationary weakly dependent time series. Those two references also establish statistical tools for that situation. For independent but not identically distributed random variables, a basic probabilistic result is in Mejzler [12]. In our EVT investigation, we start considering and supporting less constraining hypotheses than independence, such as stationarity and extremal dependences. We present tests to verify them and the guarantees that can be provided to the results of the extreme value theory with such assumptions.

3.1 Stationarity

Given execution time data sets, to verify stationarity the autocorrelation can be computed with lag plots, or turning point test can be performed. These are to model the relationship that exists between measured observations.

Table 1 Independence, stationarity and identical distribution tests.

	runs test	Kolmogorov-Smirnov test	autoregressive	Ljung-Box test
single-path	0.506	0.02804	AR(1)	0.07153
multi-path	0.6832	0.5434	AR(7)	0.2169
normal mono-variate	0.5755	0.8222	AR(0)	0.4593
multi-variate	0.729	0.4653	AR(0)	0.9825

Hypothesis testing means deciding, from a number of observations, whether one should consider a property to be true or not. The resulting ρ-value tells whether accept or reject the null hypothesis H_0. Normally, $\rho > 0.05$ validates the null H_0, while $\rho \leq 0.05$ rejects H_0, thus validates the alternative H_1.

With no mean of formalism, a process is stationary if its mean, variance and autocovariance structure do not change over time. An autoregressive (AR) model is a representation of type for random process: i) $AR(0)$ denotes the sequence of observations without dependence – white noise, ii) $AR(1)$ is process with a positive φ parameter where only the previous observation in the process and the noise term contribute to the output, and so on. We make also use of the Ljung-Box (LB) test, which examines whether there is significant evidence for non-zero correlations between lags. Large ρ-values from the LB test suggest that the series is not stationary, thus there is no trend between consecutive observations; this would support independence.

A test applied in [2] aims at proving that samples are independent by looking for randomness. This is called runs test, or Wald Wolfowitz test, where randomness is sought within the observed data series by examining the frequency of "runs"; a "run" is a series of similar responses. Furthermore, we consider the Kolmogorov-Smirnov (KS) test to verify the identical distribution hypothesis and check if the observations follow the same distribution.

Table 1 describes the hypothesis verification tests and their results with respect to the input measurements. For runs test, KS test and LB test, results are given as ρ-value. Noticeably, real execution traces are independent (single-path) or stationary (multi-path). The EVT can be applied to real cases, since their randomicity is enough for the EVT application. This means that there would not be the need for extra randomicity, as for example with random replacement cache policies, [2, 1].

3.2 Dependence of the extreme samples

While showing that just stationarity is needed to apply EVT, we can still get independence as far as extreme samples are concerned. Indeed, EVT can still be applied on time series with temporal dependence if the extreme samples are sufficiently separated in time. In that case, extreme samples can be considered as independent. To estimate the dependence level of extreme samples, it can be interesting to compute the extremogram of the samples.

An extremogram is a measure of extremal dependence for time series measurement [3]. Contrary to the usual methods for characterizing the dependence of samples, it only focuses on their extreme values. Let us firstly consider the theoretical definition of an extremogram.

The extremogram $\rho(h)$ of a stationary time series $(X_t)_{t\geq 0}$ is defined by: $\rho(h) = \lim_{n\to+\infty} \frac{P\{\mathcal{X}_0 > a_n, \mathcal{X}_h > a_n\}}{P\{\mathcal{X} > a_n\}}$, with a_n a sequence such that $P\{|\mathcal{X}| > a_n\} \approx n^{-1}$. The variable h can be seen as a correlation length.

a) Single-path time series b) Multi-path time series

Figure 2 Extremogram of single-path time series and multi-path time series examples.

In practice, if one assumes that the consecutive samples $(X_t)_{1\leq t\leq N}$ are available, an estimator $\hat{\rho}(h)$ of the extremogram $\rho(h)$ has been proposed as $\hat{\rho}(h) = \frac{\sum_{t=1}^{N-h} \mathbf{1}_{(X_t>a, X_{t+h}>a)}}{\sum_{t=1}^{N} X_t>a}$, where $\mathbf{1}_{(X_t>a, X_{t+h}>a)}$ is equal to 1 if $(X_t > a, X_{t+h} > a)$ and 0 otherwise. The threshold a is experimentally set as the 0.98-quantile of the samples.

Several remarks can be made on the estimator $\hat{\rho}(h)$. Firstly, this estimator varies between 0 and 1. When $\hat{\rho}(h) \longrightarrow 1$, the extremal samples are highly correlated. In that case, group of consecutive extremal samples can be observed in time. De-clustering algorithm is then often required in order to apply safely EVT [6]. When $\hat{\rho}(h) \longrightarrow 0$, the extremal samples are uncorrelated and can arise as individual sample in the time series. EVT can then be applied with more confidence.

Figure 2 shows the extremogram to single- and multi-path cases. The extreme samples of these 2 time series have a limited correlation, $\hat{\rho}(h) < 0.1$, and de-clustering is not required.

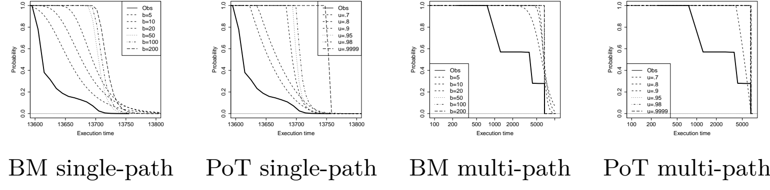
4 Extreme Value Theory approaches

The idea behind the EVT is close to a black box approach which is applied to avoid knowledge of the system, thus overcoming the complexity that today's systems have. The advantage that EVT offers to pWCET estimation is about the relatively small cost and good accuracy of the pWCET estimations. Unfortunately there exists complexity due to parameter selection. Indeed, in both BM and PoT cases, there are parameters to be defined (respectively b and u), and their impact to the resulting pWCET has to be considered. In this section we depict the differences that exist between the two forms of EVT applied to task execution time.

4.1 EVT parameters

To evaluate the impact of block size and threshold parameters we apply EVT by changing those parameters within a certain range. For BM, the size of the blocks b is such that $b \in \{5, 10, 20, 50, 100, 200\}$. For PoT, the thresholds are selected via the quantiles $q(p)$, where p is the quantile probability. Thus, it is $u \in q(0.7), q(0.8), q(0.9), q(0.95), q(0.98), q(0.9999)$. Figure 3 compares the parameters effects with four different input distributions and the 1-CDF representations. For the empiric distributions there is more accuracy from the EVT estimations, at least within a certain exceeding probability range $[1, 10^{-6}]$. While in the single-path case the PoT appears to be more accurate[7], in case of multi-path execution traces, it is BM which is more accurate. The effectiveness of the EVT depends on the shape of the input distribution, and it is not possible to conclude about one EVT approach being better than another. To note how increasing the parameters, i.e. increasing the block size or the threshold (the quantile), the quality of the EVT estimation degrades not linearly.

[7] Accuracy is empirically defined with respect to the measurements, since the exact pWCET is unknown.



BM single-path PoT single-path BM multi-path PoT multi-path

Figure 3 EVT pWCET distributions and values varying block size and threshold.

Table 2 10^{-9} exceeding thresholds with different EVT parameters compared with the measured value.

	max	BM $b = 5$	BM $b = 10$	BM $b = 20$	BM $b = 50$	BM $b = 100$	BM $b = 200$
single-path	13755	14388	14312	14184	13886	13894	13918
multi-path	6750	41812	26353	19065	6779	6820	6920
	max	PoT $p(0.7)$	PoT $p(0.8)$	PoT $p(0.9)$	PoT $p(0.95)$	PoT $p(0.98)$	PoT $p(0.9999)$
single-path	13755	13758	13758	13769	14006	14729	13752
multi-path	6750	6861	6861	6974	34563	$4.94 * 10^8$	6851
	max	$u, p(0.7)$	$u, p(0.8)$	$u, p(0.9)$	$u, p(0.95)$	$u, p(0.98)$	$u, p(0.9999)$
single-path	13755	13623	13636	13684	13695	13701	13751
multi-path	6750	3985	6697	6712	6714	6715	6832.33

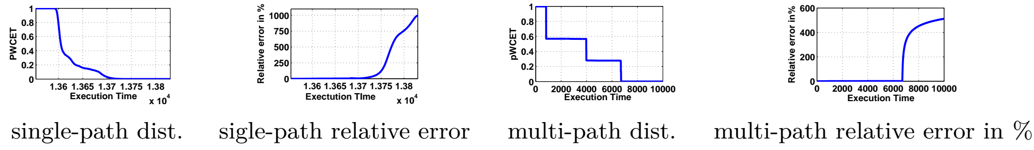
In Table 2 it considered the pWCET tuple definition $\langle C^*, 10^{-9}\rangle$. Here is understandable the difficulty that EVT has in accurately estimating the pWCET with extreme parameters, such as small b and PoT with few values above the threshold ($u = 0.9$ or more). Estimated C^* at limit cases are either very far from the measured values, i.e. multi-path with $u = 0.95$, $u = 0.98$, and $b \leq 20$, or very close to the measured values, i.e. $b = 100$ and $b = 200$. Both PoT and BM pWCET estimation remains safe with no underestimations. But while increasing b the pessimism decreases, and perhaps the capability of embedding rare events decreases, increasing u it is the accuracy of the pWCET estimation to reduce resulting into more pessimistic pWCET estimations. In case of multi-path those trends are more evident due to the multi-variate distribution the EVT has to handle. In there, interesting are the results for $u = 0.98$ and $u = 0.9999$, respectively with a huge pWCET estimation and a maybe too small one. Critical cases have to be avoided in order to PoT safe and sound. Last three rows of the table are the thresholds used by the PoT at the respective quantiles, to give an idea about where are the estimated pWCETs.

Both table and figure describe outline the complexity in selecting the best parameter, which depends on the input measured distribution.

4.2 EVT robustness: estimation of relative error

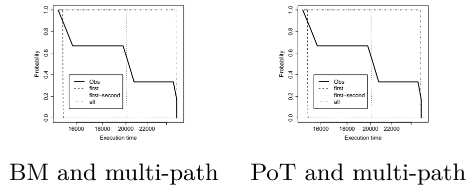
Bootstrap [4] is a well-known statistical method that enables to estimate characteristics of a statistics. It can notably be applied to estimate relative error and confidence interval of the pWCET distribution and quantiles obtained with EVT.

For that purpose, it is firstly needed to re-sample the data $X_1, \ldots, X_N$ to obtain a bootstrap re-sample $X_1^*, \ldots, X_M^*$. The term X_j^* is a sample set of size N and is determined from $X_1, \ldots, X_N$ by random sampling with replacement. From the bootstrap re-sample $X_1^*, \ldots, X_M^*$, one can then estimate M pWCET distributions, $\text{pWCET}_1, \ldots, \text{pWCET}_N$ and the associated 10^{-9}-quantiles obtained with EVT. The relative errors or the 95% confidence interval of these quantities are then easily computable.



single-path dist. sigle-path relative error multi-path dist. multi-path relative error in %

Figure 4 pWCET and associated relative of single-path and multi-path time series examples.



BM and multi-path PoT and multi-path

Figure 5 EVT path variation and frequency impact.

Let us apply bootstrap to estimate relative errors and confidence interval for single- and multi-path series with $M = 100$. The pWCET distribution and its corresponding relative error are plotted in Figure 4 for single-path series. The 10^{-9} quantile of the pWCET distribution is equal to 13766 with $[13743, 13797]$ as 95% confidence interval. The pWCET distribution and its corresponding relative error are plotted in Figure 4 for multi-path series. The 10^{-9} quantile of the pWCET distribution is equal to 6922 with $[6730, 7454]$ as 95% confidence interval. In both single- and multi-path series, the pWCET distribution obtained with bootstrap is smooth since it is estimated with a mean operator. The accuracy of the pWCET distribution decreases with execution time, indeed the probability associated to pWCET becomes very low when execution time increases. This probability is thus badly estimated since the number of samples N is constant.

4.3 EVT robustness: completeness

In this section we show some of the limits of the EVT once applied to the worst-case execution time problem. It is due to the knowledge of the system: in order to be effective (and safe), the EVT has to know which are the worst case execution conditions.

In Figure 5 we have applied the artificial multi-modal distribution as input to the EVT. The multi-modal distribution has been obtained combining three normal distributions with different mean values. In our test we have changed the execution time inputs. “first” is the measurement trace obtained from the normal distribution with the smallest mean. The EVT applied to that trace of observations is labeled “first”. “first-second” is the case where the EVT is applied to a trace obtained with the least mean and second least mean normal distributions. “all” has the whole set of observations from the three distributions. We notice that, just with partial information (not the whole multi-modal distribution but portions of it), the EVT is not able to safely infer the extremes, and thus pWCETs.

The EVT needs complete set of inputs (depicting the measurement conditions) in order to be safe: EVT robustness depends on the knowledge of the system and its execution conditions, including variability sources due to input variability and path coverage.

5 Conclusion

With this work we begin a formal analysis of the extreme value theory applied to the probabilistic worst-case execution time problem. We first present the problem as well as the EVT and its two possible approaches, i.e. block maxima and peak over threshold. Their applicability together with assumptions are verified, proving that the i.i.d. hypothesis is too strict, while stationarity and extremal dependences are allowed for safe EVT pWCET estimations. Furthermore, we provide initial verification means to EVT complexity and parameter selection, outlining the impact that parameters have on the pWCET estimation. Finally, we introduce the notion of robustness for EVT estimations.

In the future, we intend to continue in those directions aiming at listing the EVT limits and its potential. This helping the developer better choosing between measurement-base approach and static timing analysis, and perhaps combining both in an efficient hybrid timing analysis.

Acknowledgement. The work of Damien Jacquemart is financially supported by DGA (Direction Générale de l'Armement) and Onera.

References

1. Francisco J. Cazorla, Tullio Vardanega, Eduardo Quinones, and Jaume Abella. Upper-bounding program execution time with extreme value theory. In *WCET*, 2013.
2. L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *23nd Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2012.
3. R. A. Davis and T. Mikosch. The extremogram: A correlogram for extreme events. *Bernoulli*, 2009.
4. B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, NY, 1993.
5. P. Embrechts, C. Kluppelberg, and T. Mikosch. Modelling extremal events for insurance and finance. *ZOR Zeitschrift for Operations Research Mathematical Methods of Operations Research*, 97(1):1–34, 1994.
6. Christopher A. T. Ferro and Johan Segers. Automatic declustering of extreme values via an estimators. *EURANDOM report, Eindhoven University of Technology*, 2002.
7. Mark K. Gardner. *Probabilistic analysis and scheduling of critical soft real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. AAI9953022.
8. E. J. Gumbel. *Statistics of Extremes*. Columbia University Press, 1958.
9. J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
10. T. Hsing. On tail index estimation using dependent data. *The Annals of Statistics*, 1991.
11. M. R. Leadbetter, G. Lindgren, and H. Rootzén. *Extremes and Related Properties of Random Sequences and Processes*. Springer-Verlag, 1983.
12. D. Mejzler. On the problem of the limit distribution for the maximal term of a variational series. *Lvov. Politehn. Inst. Naucn Zap. Ser. Fiz.-Mat.*, 1956.
13. J. Pickands. Statistical inference using extreme order statistics. *Annals of Statistics*, 3(1):119–131, 1975.
14. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, T. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 2008.

Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art

Gabriel Fernandez[1,2], Jaume Abella[2], Eduardo Quiñones[2], Christine Rochange[4], Tullio Vardanega[5], and Francisco J. Cazorla[2,3]

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

Abstract

The real-time systems community has over the years devoted considerable attention to the impact on execution timing that arises from contention on access to hardware shared resources. The relevance of this problem has been accentuated with the arrival of multicore processors. From the state of the art on the subject, there appears to be considerable diversity in the understanding of the problem and in the "approach" to solve it. This sparseness makes it difficult for any reader to form a coherent picture of the problem and solution space. This paper draws a tentative taxonomy in which each known approach to the problem can be categorised based on its specific goals and assumptions.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Contention, Multicores, WCET Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2014.31

1 Introduction

At a conceptual level, the intent of timing analysis is to provide, at low-enough cost, a WCET bound for programs running on a given processor. Ideally, the transition to multicore processors should allow industrial users to achieve higher levels of guaranteed utilization, together with attractive reduction in energy, design complexity, and procurement costs. Unfortunately however, the architecture of multicore processors poses hard challenges on (worst-case) timing analysis. The interference effects arising from contention on access to processor-level resources in fact need far greater attention than in the singlecore case, as much greater is the arbitration delay and state perturbation that resource sharing may cause, and consequently much greater is the "padding" factor that needs to be captured in the computed bounds to compensate for the relevant effects.

From a bottom-up perspective of the system architecture, the utilization that individual tasks make of many of those shared resources at the processor level is too low to justify a dedicated use of them. Efficient use of those resources, which is needed for decent average performance, requires sharing. However, resource sharing shatters tightness in timing analysis and endangers the trustworthiness of the bounds computed with it.

Different approaches (angles) have been considered to address the timing effects of contention for shared resources. However, there is evident lack of common understanding of the problem space, in terms of processor features, and of the assumptions made to solve it.

© Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla;
licensed under Creative Commons License CC-BY

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 31–42

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

Contribution. This paper aims to contribute to the community understanding of how bounds can be computed for the execution time of software programs running on processors with shared hardware resources. Special attention is given to capturing the assumptions made by each considered technique, as a way to gage the applicability of its results. The authors' intent is to capture the principal angles of attack to the problem, as they emerge from the state of the art, flag-shipped by specifically representative strands of work, without necessarily achieving exhaustive coverage of the literature. In the authors' opinion, whereas the number of works on multicore resource contention is vast and growing, the community lacks a common understanding of the big picture, which would be needed to determine which new techniques are needed to best address the problem from the perspective of benefiting application.

2 A tentative taxonomy of state-of-the-art techniques to analyse the timing impact of resource contention

Under the umbrella term of resource contention, we capture the various forms of timing interference that software programs suffer owing to access to shared hardware resources. Notably, our analysis does not cover the contention on access to software resources. Furthermore, contentions arising from parallel execution of a software program fall outside of our analysis and are recognized in Section 4 as an important emerging ramification of the problem.

The challenge of contention in multicore processors has been addressed with various approaches. This paper classifies them in four broad categories, dependent on where they seem to direct their focus: (1) on system considerations, which address the contention problem top down, from the software perspective; (2) on WCET considerations, which take the opposite view, studying how contention phenomena affect the timing behaviour of the software; (3) on architecture considerations, which devise processor features and arbitration policies that help achieve composable timing behaviour; and on (4) Commercial Off-The-Shelf (COTS) considerations, which propose processor-specific ways to deal with processor-specific contention and arbitration features. We discuss the approaches in each category in isolation and we break them down into subgroups where appropriate.

2.1 System-centric techniques

System-centric techniques take a top-down approach to the problem. Those techniques take an off-chip, hence coarse-grained, perspective. Off-chip resources have longer latency and also a higher degree of visibility from the software standpoint than on-chip resources. For instance, at software level one can easily tell where a set of addresses is mapped to memory, but it is (much) harder to determine which data item is (or is not) in cache at a given time. In fact, the cache impact has characteristics that can be captured, from different angles and with different precision, with techniques that we classify in different categories of our taxonomy (system-centric and WCET-centric). Besides this cross-boundary overlap, the techniques in this category predominantly focus on off-chip resources.

We single out three angles worthy of specific discussion: timing analysis frameworks; access scheduling and allocation; and works on COTS architectures.

2.1.1 Timing analysis frameworks

In general, these techniques assume that on-chip shared resources (e.g. core-to-cache bus, caches, etc) are replicated or partitioned across cores, so that software programs allocated

to a core suffer no contention on access to on-chip resources. These techniques also assume that analysis frameworks model off-chip shared resources in isolation and provide worst-case access timing bounds for them. The impact of contention is only considered for the off-chip resources and it is captured compositionally[1], when the WCET of the software program, determined assuming no contention, is increased by delay factors that consider the sources of off-chip contention in the presence of co-runners.

The shared resources considered in this class of approaches are assumed to process one request at a time. It is also assumed that the corresponding services cannot be pre-empted (or split or segmented). It is further assumed that the requests are synchronous so that the requesting task is stalled while the access request is served. In reality, some requests are asynchronous not blocking the calling task execution. The analysis focuses on individual tasks, whose program units are logically divided into blocks for which maximum and minimum access bounds and execution time bounds are derived.

For approaches in this class, the access to the shared resource is assumed to be arbitrated by either a TDMA bus [9] or a dynamic arbitration bus [29] or else an adaptive bus arbiter [10]. For TDMA buses, focus is on determining the worst-case alignment of the requests in the TDMA schedule. As the bus schedule is static, co-running tasks do not affect one another's execution time, which makes their execution time composable with respect to the bus. The fact that service is assumed run-to-completion and that requests do not overlap simplifies the problem.

For dynamic arbiters, the workload that a task places on the shared resource affects the access time of the co-running tasks, which breaks time composability. This type of arbiters has generated a research line of their own. Several authors [4, 29] propose how to derive bounds for the number of accesses per task in a given period of time. The timing analysis for an individual task therefore depends on the request workload generated by the co-runners in that time duration. Interestingly, while the number of accesses that a task generates to the resource can be considered independent of the co-runners as long as caches are partitioned, the task's frequency of access depends on how often the co-runners delay the task's requests. The cited models capture that dependence.

Adaptive arbiters (as in, e.g., FlexRay) exhibit a window with per-requester slot scheduling combined with a window in which requests are dynamically arbitrated; this trait makes them show characteristics that we have seen above as distinctive for static and dynamic arbiters.

The authors of [30] provide a useful survey of how time-deterministic approaches to bus arbitration and scheduling for multicore processors can be captured, compositionally, by timing analysis techniques. The cited work also presents benchmark-based empirical evidence of the degradation that TDMA arbitration causes to average-case performance in comparison to other techniques with acceptable characteristics in terms of time determinism.

2.1.2 Task scheduling and allocation

The state-of-the-art approaches to multicore scheduling and schedulability analysis that match the techniques which fall in this category can be grouped in two sets: those that ignore contention issues at their level and leave it to WCET analysis; and those that consider

[1] We use the term compositional to signify that some property of an individual part of the system can only be determined on (assumed) knowledge of the constituents of the system. This is in contrast with the term composable, which regards those properties of an individual part that can be determined considering that part in isolation and hold true on composition into the system [28].

it as a factor of influence to task allocation, which is adjusted to attain increased schedulable utilization.

There essentially exist three classes of scheduling algorithms, which differ in the way they assign tasks to cores [5]. Partitioned and global scheduling fall on the respective extremes of the spectrum: the former statically maps tasks to cores, so that a task can only be scheduled on the core it has been assigned to; the latter allows tasks to migrate jobs from one core to another and does not leave any core idle if there is some work to be done (work conserving), at the expense of possibly costly task (job) migrations. The former risks considerable under-utilization of the processing resources, especially for tasks with medium to high loads. The middle of the spectrum is occupied by clustered or semi-partitioned algorithms, which – with various techniques and for different goals – only allow or cause statically-set groups of tasks to migrate within statically or dynamically determined subsets of cores.

Contention oblivious. The principal works on scheduling algorithms and associated schedulability analysis for multicore processors assume that the WCET of all tasks is given in input. Thus, they assume that the WCET bounds may be determined before decisions are made on task mapping to cores and on scheduling at run time. This is tantamount to postulating that the WCET bounds are composable, that is to say, free from variations determined by the presence of contenders in the system. Ironically, the only plausible way in which WCET bounds can actually be made to be composable for use in schedulability analysis for multicore processors is by increasing them compositionally, by a factor determined by given patterns of conflicts that upper bound the actual contention delays suffered at run time. In essence, the approaches of this kind escape the intrinsic (and painful) circularity between the dependence of WCET analysis on knowledge of the contenders and the dependence of schedulability on knowledge of the WCET of the tasks in the system, by inflating the WCET budgets so that they can always be trusted to upper bound the actual costs.

Contention aware. Techniques such as [32, 12] focus on the shared last-level cache as one of the main resources in which contention occurs. The cited works benefit from hardware proposals that split the cache into different ways or allocate program data into different pages (colors) so that each task is limited to use a subset of the sets in cache, thereby reducing conflicts.

These works often assume partitioned scheduling for software programs, so that conflicts can be determined in a less pessimistic way, and focus their attention on devising cache-aware allocation algorithms that consider the mapping of tasks to cores determined by partitioning. Some of the works focus on how to assign colors (i.e. set partitions) to the tasks. It is also the case that works in this class do not address the contention occurring in other shared resources like the memory.

Other works [23] build on hardware proposals that control the interaction in several hardware resources (e.g. on-chip bus, cache and memory controller) in addition to the cache. These proposals also consider task allocation and scheduling.

2.2 WCET-centric techniques

WCET-centric techniques determine the impact of contention in the access to shared resources as part of WCET analysis. For multicore architectures, shared resources include cache memories, buses and memory controllers, but some approaches have also been designed to support intra-core resource sharing (e.g., pipeline and functional units in multithreaded

cores [3]). The objective of WCET-centric techniques is to derive safe stall times that can be accounted for at instruction-level timing analysis. We distinguish between the approaches that consider all the competing threads/tasks together to exhibit the possible interleavings of their respective accesses to the shared resource, from those that exploit a static allocation of slots among cores. In the latter category, some contributions include WCET-based strategies to optimize the mapping/scheduling of threads/tasks to cores to optimize the global WCET and/or to improve schedulability.

2.2.1 Joint analysis of concurrent tasks/threads

One way to identify how contention may impact the WCET of a task is to combine the analyses of concurrent tasks to identify where they can interfere. Two kinds of interference are considered here: spatial (tasks share storage, e.g. a cache) and temporal (tasks share bandwidth, e.g. a bus). Both incur additional delays.

Techniques that address spatial contention start by perform individual tasks analyses, then determine how contention affects their results. More precisely, they determine which cache lines used by one task might be replaced by another task in a shared L2 instruction [14][33] or data [11] cache. The analysis of contention does not account for the exact respective timings of tasks (then could be valid for any schedule, provided all possible concurrent tasks are known at analysis time). However, [33] improves the accuracy of the analysis by considering constraints on task scheduling (non-preemptive, priority-based, with task inter-dependencies), which allows bounding tasks lifetimes and limits the opportunities for contention.

To account for temporal conflicts and derive instruction timings, possible interleavings of (statically-scheduled) threads must be explored. Several approaches use timed automata to represent both the tasks and the state-based behaviour of hardware components. All these automata are combined and model checking techniques are used to determine the WCET though a binary search process. [6] focuses on the shared L2 cache with fixed cache miss latency. A shared bus with First-Come-First-Served (FCFS) or TDMA arbitration is analysed in [19]. The weakness of these approaches is in the huge number of states to be handled.

2.2.2 Independent analysis of tasks/threads

Some techniques leverage the deterministic guarantees offered by the underlying hardware on access to a shared resource. Thanks to such guarantees, they can analyse the WCET of one task/thread independently of the concurrent workload.

The impact of arbitration delays on a TDMA bus with uniform slot size is explored in [31]. The cited work presents an approach to evaluate the misalignment of accesses with TDMA slots (TDMA offsets). A TDMA-composable system is assumed: arbitration delays neither impact instructions that do not access the bus nor the bus access time (except for the arbitration delay).

Some of the hardware solutions to enforce access guarantees do not offer equal opportunities to all threads. Cache partitioning techniques may allocate partitions with different sizes [23]. Bus arbiter may grant a different number of slots to each core, as in [18] or [26]. Those techniques use these mechanisms to increase the performance achievable by combined task-to-core allocation and scheduling decisions, especially in the case of unbalanced workloads (with variable demand levels to the shared resource). Performance benefits are obtained as a result of reducing the WCET bound predictions for the affected tasks. As we

noted in Section 2.1, our taxonomy is not clear-cut enough to place some of these techniques uniquely in one class, as they might arguably also belong to the system-centric group.

2.3 Architecture-centric techniques

Several hardware design paradigms have been proposed to deal with the inter-task interference caused by contention for shared hardware resources. Four topical approaches can be singled out in this group: the time-triggered architecture [36]; PRET [37]; CompSOC [8]; and MERASA [38].

One of the differentiating elements for these approaches is whether they achieve composability at the level of the WCET bounds that they allow computing or at higher levels of abstraction. The objective of the former solution is to support determining WCET bounds for individual tasks in isolation, independently of the activity of their co-runners. When that is guaranteed, the execution time of a task may well suffer variations caused by contention effects caused by some of its co-runners, but its WCET estimate stays valid. With the latter type of solutions, composability is achieved by regulatory mechanisms operating at run time, and thus with effect on the task execution time. Those regulatory mechanisms ensure that the activity of the co-runners cannot affect the response time of the hardware shared resources. This form of composability may place more requirements on the processor hardware than the former approach. In general it requires that the access time to a hardware shared resource stays always the same irrespective of the actual load of the system. To that end, a resource that might respond ahead of time is stalled until the agreed latency for the request is reached.

From another angle, it is worth noting that a trade off arises as a consequence of the observation that the pursuit of time composability always comes at the cost of some (over-provisioning) pessimism. The effect of this (static) over-provisioning allows tightening the WCET bounds, because they eradicate sources of variations, but at the cost of renouncing the true meaning of time composability (as independence from the presence of contenders), which is central to the incremental verification needs of integrated architectures such as Integrated Modular Avionics (IMA).

Somewhat orthogonal to the discussion above, the focus of several proposals is to upper bound the access time to hardware shared resources, either indirectly, by guaranteeing pre-determined bandwidth on access to the resource, or directly by ensuring bounds on the access time (comprised of the wait time preceding access upon request, and the actual service time).

The techniques of interest from this angle vary for *stateless* and *stateful* resources. Stateless resources have an access time that is not or only very modestly dependent on execution history. A single-cycle latency bus is a typifying example of resources of this kind. If the bus had a two-cycle latency, then the service time of a request might depend on whether the preceding request was sent the cycle before the current one gets ready. Caches are a difficult exemplar of stateful resources. This is because the state-dependent effect builds up with history of execution, which causes analysis to have to keep track of the full history of access. Truncated information requires conservative assumptions to be made. This difficulty explains why the typical solution proposed for caches consists in splitting its space in small areas assigned to individual tasks, so that history becomes much smaller (and free of conflicts with co-runners) and thus easier to trace. This can be done dividing the cache into different banks or different ways [22].

The most prominent stateless resources on which the real-time community has focused are network-on-chip (NoC) and memory controllers. For the interconnection network, proposals

exist which range from from simple buses [35] or rings [20] to more complex solutions such as those described in [41]. All share the goal to provide some type of bound to the longest time a request has to wait to get access to the resource. For the memory controller, proposals with the same goal exist [1, 21], though the actual solutions are more complex since the state retention is higher.

2.4 COTS-based techniques

The goal of several works focusing on real hardware is to analyse how amenable a given multicore design is for real-time analysis. To that end authors analyse different shared resources as well as their impact on execution time. For many resources the manuals of the processor under analysis rarely provide all the required information to analytically derive those bounds. As a result, the way in which the authors derive bounds is different from previous approaches and it is based on experimentation on the specific architectures under analysis [25, 39, 17]. These works include analysis of the FreeScale P4080 and some FPGA versions of the Aeroflex Gaisler LEON4.

Another set of works is carried out at an analysis level providing understanding of the timing behaviour of hardware shared resources and the challenges they bring to timing analysability [13, 27, 42]. Finally, some of the works on software-cache partitioning (page colouring) have been done for processors like the ARM Cortex A9 [12].

3 Critique

This section reviews the techniques captured in the taxonomy presented in Section 2 against multiple criteria including: (1) the presence of overlaps between them; (2) the presence of gaps among them; (3) the realism of their assumptions; (4) the challenges in taking that technique to industrial use; and (5) the relation between the confidence on the bounds determined by timing analysis and the assurance guarantees suitable for the application domain.

Much like the proposed taxonomy, the review discussed here is not meant to be exhaustive. It therefore does not cover all criteria for all techniques. Instead, it only aims at singling out specific problematics that we consider to need particular attention by the prospective user and further study by the research community.

System-centric techniques. The principal limitation with this class of techniques stems from their resting on two strong assumptions: that programs can be statically subdivided into (super)blocks for which bounds on resource usage can be derived; and that only one shared resource needs attention, which also does not support split transactions. The former restricts applicability to programs that can be divided into blocks for which maximum and minimum access bounds and execution time bounds can be derived. Further, working at block level may increase pessimism because every block is given a single worst-case cost value, which may be higher than the actual cost in the worst-case traversal of that block as taken by the program. The latter assumption reduces the applicability of the solution against increasingly common hardware.

For dynamic arbiters, the critical factor is in the dependence of their timing analysis on the request workload generated by the co-runners of the program of interest in a given time duration. On the one hand this trait reduces pessimism since the durations in which conflicts on access may occur can be better determined. On the other hand, it breaks time composability and resorts to compositionality. The latter defect may be a serious impediment

to incremental verification, which is a prerequisite to high-criticality domains (e.g., avionics). Budgeting in advance for the co-runners is obviously one countermeasure to that, but at the direct cost of over-provisioning.

WCET-centric techniques. The main challenge for this class of techniques arises from having to find tractable ways to analyse increasingly complex hardware. The abstract interpretation approach on which those techniques rely is inherently exposed to the state explosion problem, which is dramatically worsened by the way in which the architecture of modern processors cause the timing behaviour of several resources to exhibit possibly large jitter, extremely sensitive to the history of execution [42]. This dependence obviously cumulates bottom-up and manifests in very complex ways at software level.

As an example we consider a TDMA bus, whose timing behaviour is easy to model with three main parameters: window size, number of contenders, and slot size per contender. Interestingly, the state space for even such a simple model is already not negligible: when the exact time of an access request cannot be determined in fact, a conservative assumption must be made on when access will be granted (which inflates pessimism) or multiple candidate access times are considered, which causes multiple states to be contemplated upward in the analysis. As more complex NoC architectures are adopted by modern multiprocessors, more parameters will be needed to model the sources of contention, with inordinate increase in the complexity and cost of the analysis tools.

Architecture-centric techniques. A recurrent question on the viability of the techniques in this class is whether the hardware design that they propose in the intent to favour time analysability, will ever reach the market. This is a question of economics that equally applies to all research domains that propose new hardware architectures. However, it is especially important to the real-time systems domain, which holds a tiny niche of the market size, in comparison to consumer products, without sufficient critical mass to swing the prevailing design criteria from optimized for the average case to well-behaved in the worst case.

This is a long-known challenge for the real-time systems community. Fortunately, perseverance and authority have shown able to win some battles, so that some of the proposed designs (e.g., cache partitioning) are indeed retained in real processors. Our view here is that the changes proposed for the bus and the memory controller are simple enough so that they can be implemented in production with moderate effort and cost, for tangible benefits on timing analysability. Whether or not that will actually happen remains to be seen.

In general all hardware approaches assume processor designs without timing anomalies. It is interesting wondering, whether processor can be made simple enough to assure freedom from timing anomalies, without this causing detriment to the attainable performance. Architectural solutions will have to be devised that combine those two objectives harmoniously, which is not the case yet with the dominant approaches to multi- and manycore processor architectures.

COTS-based techniques. The techniques that belong in this class face the challenge that the architectural properties needed to provide full time isolation or time predictable interaction among processor cores cannot be had owing to the lack or inaccuracy of specification information or IP restrictions. Various approaches have been proposed to live with the consequent uncertainty, which all require building confidence arguments that accord with the requirements and practices of the application domain. The work in [34] makes an interesting review of how safety assurance guarantees relate to stipulating bounds on execution time.

4 Other aspects of interest

In this section we briefly touch upon two other aspects that, for different reasons, are tangent to questions addressed in this paper. One aspect, parallel programming, intrinsically enabled and called for by multicore processors, presents a novel, emerging challenge to bounding contention effects. The other aspect, with interesting potential and important ramifications, stems from shifting the angle of attack to the timing analysis problem, from finding a single value, the smallest possible computable upper bound, for all possible executions of a software program, to determining a probability function whose tail can be cut at the exceedance threshold of interest to the system.

Parallel applications. Communication of data in message-passing and synchronisations in shared-memory programming induce delays that must be accounted for in execution times. The focus is on deriving the WCET of the longest thread.

Two kinds of synchronization exist. (1) Mutual exclusion is very similar to accessing a hardware shared resource that can serve a single thread at a time. Computing the worst-case stall time of a thread at a critical section is analogous (when threads are served in a FIFO order) to computing the worst-case delay to a round-robin bus [15]. Stall times can then be integrated to instruction-level timing analysis. Another approach is to use timed automata and a model checker, as in [6]. In [7], a shared-memory parallel programming language is introduced and a fix-point analysis is able to identify all the possible thread interleavings at critical sections. (2)Progress synchronisation includes barriers as well as condition signalling and blocking message passing. Collective synchronisations (barriers), where all threads meet, are easier to consider since the goal is to compute the WCET of the longest thread, i.e. the last one to reach the barrier [15]. For point-to-point synchronisations (condition signalling or message passing) however, stall times depend on the respective progress of the threads. In [40], parallel applications where threads communicate through message passing are considered. A joint analysis is proposed, where the analysis of worst-case communication times is integrated into the analysis of the global WCET. The approach consists in merging the control flow graphs of parallel threads, then adding edges to model the synchronisations (dependencies) related to sending/receiving messages.

Some system-centric approaches have been extended to parallel fork-join applications and decide altogether the allocation of threads' memory in caches, the scheduling of threads' accesses to the shared bus and the scheduling of the threads themselves to the cores [2].

The probabilistic approach. Timing analysis techniques can be broken down into *deterministic*, which produce a single WCET estimate, and *probabilistic* that produce multiple WCET estimates with associated exceedance probabilities. It is noted that our discussion above has focused on standard (deterministic) timing analysis techniques. While both deterministic and probabilistic approaches try to reach time predictability, the former do so by advocating for hardware and software designs that are deterministic in their execution time, while the latter advocates for hardware and software designs that have a randomized timing behaviour, to produce WCET estimates that can be exceeded with a given *probability*.

The probabilistic approach deals with contention by means of time-randomised bus arbitration policies [16] as an alternative to deterministic policies such as round robin. Similarly, in [24] it is proposed a time-randomised shared cache for which impact of contention among co-running tasks can be determined. The main feature of this cache is that it does not split the cache, either into ways or sets, to prevent the interaction among co-running

tasks. Instead, it controls how often tasks evict data from cache as a way to bound the impact of contention on tasks' WCET estimates.

5 Conclusions

A wealth of relevant literature addresses the problem of finding a bound on the timing effect of contention on access to hardware shared resources in modern multicore processors. The industrial practitioner, and the researcher alike, who approach that body of knowledge without a preconceived solution in mind, may have serious difficulties in seeing the "big picture" of what options are possible and at what consequences. This paper sketches an initial taxonomy of the principal approaches that appear in the state of the art, and discusses gaps and overlaps among them.

Acknowledgements. The research leading to this work has received funding from: COST Action IC1202, Timing Analysis On Code-Level (TACLe); and the parMERASA and PROXIMA grant agreements (respectively no. 287519 and 611085 from the Seventh Framework Programme [FP7/2007-2013]). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence.

References

1 B. Akesson et al. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.
2 A. Alhammad and R. Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *DATE*, 2014.
3 P. Crowley and J.-L. Baer. Worst-case execution time estimation for hardware-assisted multithreaded processors. In *HPCA-9 Workshop on Network Processors*, 2003.
4 D. Dasari and V. Nelis. An analysis of the impact of bus contention on the WCET in multicores. In *HPCC-ICESS*, 2012.
5 R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), 2011.
6 A. Gustavsson et al. Towards WCET analysis of multicore architectures using UPPAAL. In *Workshop on WCET Analysis*, 2010.
7 A. Gustavsson et al. Toward static timing analysis of parallel software. In *Workshop on WCET Analysis*, 2012.
8 A. Hansson et al. Compsoc: A template for composable and predictable multi-processor system on chips. *TODAES*, 2009.
9 A. Schranzhofer et al. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.
10 A. Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.
11 B. Lesage et al. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *RTNS*, 2010.
12 B. Ward et al. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.
13 D. Dasari et al. Identifying the sources of unpredictability in COTS-based multicore systems. In *SIES*, 2013.
14 D. Hardy et al. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

15 H. Ozaktas et al. Automatic wcet analysis of real-time parallel applications. In *Workshop on WCET Analysis*, 2013.
16 J. Jalle et al. Bus designs for time-probabilistic multicore processors. In *DATE*, 2014.
17 M. Fernández et al. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT*, 2012.
18 M.-K. Yoon et al. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. *RTSS*, 2011.
19 M. Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS 2010*, 2010.
20 M. Panić et al. On-chip ring network designs for hard-real time systems. In *RTNS*, 2013.
21 M. Paolieri et al. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.
22 M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.
23 M. Paolieri et al. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS '11*, 2011.
24 M. Slijepcevic et al. Time-analysable non-partitioned shared caches for real-time multicore systems. In *DAC*, 2014.
25 P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.
26 R. Bourgade et al. Predictable two-level bus arbitration for heterogeneous task sets. In *ARCS*, 2013.
27 R. Wilhelm et al. Designing predictable multicore architectures for avionics and automotive systems. In *Workshop on Reconciling Performance with Predictability (RePP)*, 2009.
28 S. Hahn et al. Towards compositionality in execution time analysis–definition and challenges. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.
29 S. Schliecker et al. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, 2010.
30 T. Kelter et al. Evaluation of resource arbitration methods for multi-core real-time systems. In *Workshop on WCET Analysis*, 2013.
31 T. Kelter et al. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 2013.
32 X. Zhang et al. Towards practical page coloring-based multicore cache management. In *EuroSys*, 2009.
33 Yan Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.
34 P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Workshop on Mixed-Criticality Systems*, 2013.
35 J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.
36 H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1), Jan 2003.
37 I. Liu et al. A PRET architecture supporting concurrent programs with composable timing properties. In *44th ACSSC*, 2010.
38 MERASA. *EU-FP7 Project: www.merasa.org.*
39 J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.
40 D. Potop-Butucaru and I. Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In *Workshop on WCET Analysis*, 2013.

41 M. Schoeberl et. al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *NoCS*, 2012.

42 R. Wilhelm and J. Reineke. Embedded Systems: Many Cores – Many Problems. In *SIES*, 2012.

On Static Timing Analysis of GPU Kernels

Vesa Hirvisalo

Aalto University
Espoo, Finland
vesa.hirvisalo@aalto.fi

Abstract

We study static timing analysis of programs running on GPU accelerators. Such programs follow a data parallel programming model that allows massive parallelism on manycore processors. Data parallel programming and GPUs as accelerators have received wide use during the recent years.

The timing analysis of programs running on single core machines is well known and applied also in practice. However for multicore and manycore machines, timing analysis presents a significant but yet not properly solved problem.

In this paper, we present static timing analysis of GPU kernels based on a method that we call abstract CTA simulation. Cooperative Thread Arrays (CTA) are the basic execution structure that GPU devices use in their operation that proceeds in thread groups called warps. Abstract CTA simulation is based on static analysis of thread divergence in warps and their abstract scheduling.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Parallelism, WCET

Digital Object Identifier 10.4230/OASIcs.WCET.2014.43

1 Introduction

In this paper, we study the timing of data parallel kernels executed on SIMT machines. The SIMT execution model (Single Instruction Multiple Threads) is typical for the abundant GPUs (Graphics Processing Units) that have become the main platform for massively parallel computing. In addition to their original purpose, graphics, GPUs are used as general purpose computing devices for applications covering a wide range from super computing to ordinary desktop computing.

GPUs are used also in some real-time applications, but their wider application for such purposes is limited by the lack of solid timing analysis methods. Especially considering safety critical applications, methods giving run-time guarantees are a must. Many embedded systems found in cars, airplanes, medical instruments, etc. are safety critical. As traditional single-core processors have ceased to scale effectively in their performance, multicore and manycore processors are needed in order to cope with the increasing computational demands of novel applications.

In this paper, we consider the timing of kernels executed on SIMT machines and focus on worst-case execution time (WCET) analysis. We make the following contributions:

- we present an abstract model of SIMT execution that is suitable for WCET analysis,
- a static WCET analysis method for SIMT executed data parallel kernels, and
- a WCET analysis of an example kernel that captures typical GPU computation.

We have structured our presentation in the following way. We start with a background section that describes GPU computation. In our background description, we use a short example kernel that we use also as a running example through the paper. We continue

© Vesa Hirvisalo;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 43–52

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

by describing our method in a section that is divided into subsections according to the main phases of our method: static divergence analysis, abstract warp construction, and the abstract CTA simulation itself. After that, we present an abstract GPU machine that we call mini-SIMT and present how our example program can be executed on the machine. This is followed by our WCET analysis applied to the program. We end our paper with a short review of related work and our conclusions.

2 Background

Graphics Processing Units (GPU) can be used as accelerators for general purpose computing in a setting that is is called GPGPU computing. GPUs are not stand-alone devices. They need a host computer to operate.

The basic operation in GPGPU computing consists of normal thread execution on the CPU of the host computer. Computationally heavy operations are *launched* to the GPU to be executed. Launches follow the abstract architecture of GPU devices, which forms the basis of GPGPU programming as exemplified by CUDA and OpenCL. The programs in such languages include code for both the host CPU and the accelerator GPU [7, 6].

Abstractly, GPUs consist of several processors with several cores each. Each of the processors has a fast local memory that its cores share. The processors communicate with each other by using a slow and large global memory.

GPU launches consist of a massive number of threads that are explicitly partitioned into blocks by the programmer. The processors on a GPU execute the blocks in any order and without synchronization until no blocks remain unexecuted. We will ignore the block structure of GPGPU programs in the rest of this paper. A broader view can be found, e.g., in [1].

2.1 Kernels

The GPU side code of GPGPU programs consist of *kernels*. A kernel defines the computation done by a single thread during a launch. The kernels belonging to the same block operate in a shared memory. For simplicity, we assume a launch to consist of a single block in the following and the GPU to consist of a single processor.

`TriangleSum` (Listing 1) is an example of a kernel. It is somewhat artificial, but it contains in a compact form many of the properties that are typical for GPGPU programming. The kernel is written in a language resembling the C language variant used in OpenCL.

To use the kernel, it must be launched. Consider, for example, a 16x16 matrix `m` to be processed by the kernel into a triangle column sum vector `v` of length 16. On the CPU side we would execute the code in Listing 2 to make a launch.

The launch causes 16 threads to be started on the GPU accelerator as indicated by `size` between the angle brackets that mark the call as an accelerator call. The 16x16 matrix `m` is in the row major layout.

Inside the kernel, we have a special variable `Tid` that is defined by the system as the number of the thread executing (e.g., if thread 0 evaluates `Tid`, it gets the value 0). As `i` is initialized to value `Tid` and incremented by `c` (i.e., 16), the reference `m[i]` will cause each thread to access the element (Tid, j) of the matrix, where j is the iteration number.

As can be seen from the code, different threads execute different number of iterations of the for loop. Only the upper right triangle of the matrix gets accessed. Further, only every second row will be summed because of the condition `d % 2` and an adjustment will be added to some elements because of the condition `d % (Tid + 1) == 0`.

Listing 1 An example kernel.

```
__kernel TriangleSum(float* m, float* v, int c) {
  int d = 0;                /* each thread has its own variables */
  float s = 0;                   /* s is the sum to be collected */
  int L = (Tid + 1) * c;
  for (int i = Tid; i < L; i += c) {
    if ((d % (Tid + 1) == 0)
      s += 1;
    if (d % 2)
      s += m[i];
    __syncthreads();        /* assuming compiler support for this */
    d += 1;
  }
  v[d-1] = s;
}
```

Listing 2 A launch of the example kernel.

```
size = 16;
float m[size] = {0, .., 255};
float v[size];
TriangeSum<size>(m, v, size);
```

By default, the threads are free to proceed at their own pace. However, to synchronize the threads the programmer has added a barrier. All active threads will wait each other at `__syncthreads()` and proceed only after all of them have reached the barrier. After the loop, each thread writes the adjusted sum of one triangle column to the result vector `v`.

2.2 SIMT execution model

GPU devices use the Single Instruction Multiple Threads (SIMT) execution model. In the model, threads are grouped into warps of constant size. For example, with the warp size 8, our example launch would consist of two warps: one for threads 0...7 and the other for threads 8...15.

In the SIMT model, each warp is processed by executing the same instruction for all of its threads. In this respect, the SIMT model resembles typical vector computing (the SIMD model). For example, the addition `s += m[i]` is a single instruction but does eight operations in parallel.

However, all threads have their own program counter. Thus, the processing of threads can differ if there is branching. In the SIMT execution model, the different progress of threads is called *thread divergence*.

Simple if statements can be handled by predicated execution. For example, the value of the if statement condition `d % (Tid + 1) == 0` can be computed in parallel for all the threads in a warp and a corresponding execution mask can be formed. The following addition statement has effect only for the threads marked active by the mask.

Nested branching, such as nested if statements, is more complicated to handle. For example, the code in Listing 3 has two nested divergences. Because of `p1`, some threads are executing `s3`. The others are split by `p2` to execute the different subbranches.

To handle such structures, stack-based reconvergence mechanisms are typically used by GPU hardware. Let the condition p1 to evaluate to the vector (1, 1, 0, 0, 0, 0, 1, 1). Thus,

Listing 3 Nested branching

```
if (p1)          /* A */
  if (p2) s1     /* B */
  else s2
else s3          /* C */
...              /* D */
```

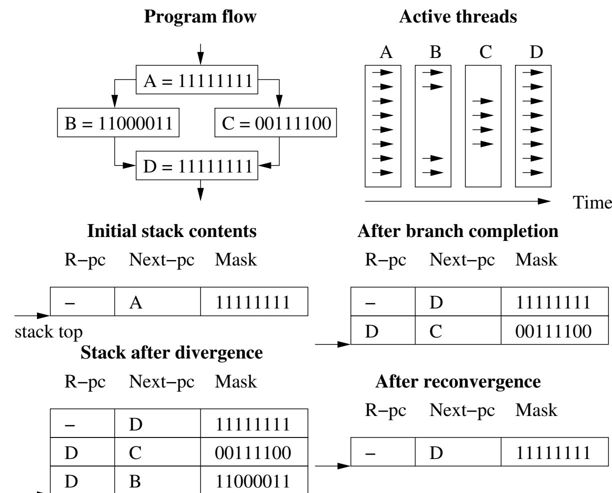


Figure 1 Thread divergence handling.

threads 3...6 should evaluate the else branch labelled `C`. The GPU hardware pushes the mask 00111100 onto a reconvergence stack together with the PC value `C` and reconvergence PC value `D`. The reconvergence PC value is needed to mark the instruction at which the divergent threads meet. The processing sequence is illustrated in Figure 1.

2.3 Avoiding memory latencies

Execution can be stalled by an instruction reading memory. GPU hardware hides such latencies by allowing multiple warps to be run concurrently. All the warps from a single block are collected into a Cooperative Thread Array (CTA). The scheduling hardware of the GPU keeps track of ready warps of a CTA, i.e., warps that are not stalled because of memory accesses (or other reasons). Typically round-robin scheduling policy is used and the scheduler is able to keep in pace with instruction issuing.

Considering our example program, the benefit of round-robin scheduling can be easily seen. By default, the warps get the same amount of execution cycles from the processor. The warps executing `TriangleSum` will progress almost with the same speed through the code and will also reach the synchronization barrier `__syncthreads()` almost simultaneously at each iteration. Thus, the barrier wait time is not very large.

We define *occupancy* as the number of ready warps. The memory latency hiding is dependent on the occupancy of the processor. As the warps may be partially filled, occupancy may differ from thread utilization.[1]

[1] Occupancy is usually defined as the percentage $actual_warps/max_warps$ (but we will use the warp count) and current hardware can support 64 warps of 32 threads (but we will use modest numbers).

3 WCET estimation method

In describing our WCET estimation method, we concentrate on features related to the SIMT execution model and omit the other features (e.g., pipelines and caches). We define the total time spent in execution as

$$T_{exec} = T_{instr} + T_{stall}$$

where T_{instr} is the number of cycles spent executing instructions and T_{stall} is the number of cycles spent in instruction execution stalls because of local memory waits.

Even assuming a simplistic hardware model, T_{instr} cannot be directly counted from the code, because the SIMT execution model allows divergence. Considering an if-else structure, the execution time is

$$T_{if_else} = \begin{cases} T_{true_branch} & \text{if all threads converge to true} \\ T_{false_branch} & \text{if all threads converge to false} \\ T_{false_branch} + T_{true_branch} & \text{if threads diverge} \end{cases}$$

Considering loops, the execution time of a warp is the time of the longest thread in the warp. Thus, for control structures, the execution time is dependent on divergence. To estimate the WCET statically, we need static divergence analysis.

Warp scheduling hides the memory latencies. In the worst case, all warps execute a memory read on consecutive cycles and the stall is

$$T_{stall} = \max(0, T_{memory} - N_{warps})$$

where T_{memory} is the memory access latency and N_{warps} is the warp occupancy. Note that T_{stall} is directly added to the total execution time, not to individual warps.

To get tight timing estimates for SIMT executed programs, we must be able to statically estimate occupancy. We base the estimation on a method that we call abstract CTA simulation. Abstract CTA simulation needs abstract warps to be constructed.

In the following, we will first describe divergence analysis, then abstract warp construction, and finally abstract CTA simulation.

3.1 Static divergence analysis

We base our static divergence analysis on GSA (Gated Single Assignment). It augments programs with value chaining information and resembles SSA (Static Single Assignment). Instead of the ϕ-function of SSA it uses three special functions to build the chains:

- γ function is a join for branches. $\gamma(p, v_1, v_2)$ is v_1 if the p is true (or else v_2).
- μ function is a join for loop headers. $\mu(v_1, v_2)$ is v_1 for the 1st iteration and v_2 otherwise.
- η is the loop exit function $\eta(p, v)$. It chains a loop dependent value v to loop predicate p.

GSA allows control dependencies to be transformed into data dependencies whose chains we can follow. We say that a definition of a variable is *divergent* if the value is dependent on the thread. If there are no divergent definitions for a branch predicate, we know the branch to be non-divergent. The details of the method can be found in [3].

3.2 Abstract warp construction

For static WCET estimation, we need timing information of the warps to be executed. We do this by constructing abstract warps. Abstract warps resemble the traditional control flow graphs. An *abstract warp* $A = (V, E)$ is directed graph. The nodes V have three node types:

- Time nodes describe code regions with two values. T_{instr} is the upper bound of the instruction execution time consumed. T_{shift} is the upper bound of the variation of the instruction execution time caused by thread divergence.
- Memory access nodes that mark places where memory access stalls may happen.
- Barrier nodes that mark places where barrier synchronization must happen.[2]

An abstract warp is constructed from the code in a recursive bottom-up way by applying:

procedure construct(program element P)
  case P
    is compound statement S_1, S_2: R = Join(construct(S_1), construct(S_2))
    is if statement for S_1, S_2: R = LUB(construct(S_1), construct(S_2))
    is loop statement with body S: R = Cons_loop_edge(construct(S))
    is memory read statement S: R = Cons_flow_edge(Time(S), Memory())
    is memory barrier statement S: R = Cons_flow_edge(Time(S), Barrier())
    is some other S: R = Time(S)

where R is the return value. The constructor Join merges S_1 and S_2 by summing the T_{instr} and T_{shift} values, if they are statically resolvable. The constructor LUB merges S_1 and S_2 by selecting the worst-case T_{instr} and the T_{shift} values, if they are statically resolvable. The constructors Time, Memory, and Barrier construct simple nodes.

3.3 Abstract CTA simulation

We use abstract CTA simulation to get execution time estimates for the kernels whose structure we can statically resolve. Instead of exhaustively executing all warps with thread masks, convergence mechanisms, and all possible scheduling interleaving choices, abstract CTA simulation considers a single abstract warp. It uses static estimation to understand the effects of multiple warps, thread divergence, memory latencies and scheduling choices.

Our abstract CTA simulation assumes the hardware to use round-robin scheduling for warps. This means that without divergence between the warps, the warps will be executed as *convoys*. A convoy is the execution of the same instruction by all warps in a single round-robin cycle. We define divergence from this scheduling as *convoying shift* T_{SHIFT}, which is the program counter distance among threads within a single round-robin cycle.

The WCET calculation is based on cumulative sum of instruction execution and memory stall times during the abstract CTA simulation. In the simulation, we use bounds for occupancy: N_{low} is the lower bound and N_{high} is the upper bound.

procedure simulate(abstract warp A, warp count WC)
  $T_{WCET} = 0$
  $N_{low} = N_{high} = WC$
  proceed through A = (V, E) **until** termination
    let $v \in V$ be the current node
    case v
      is time node:
        $T_{WCET} \mathrel{+}= N_{high} * T_{instr}$
        $T_{SHIFT} \mathrel{+}= T_{shift}$

[2] Note that barriers can cause deadlocks in actual programs. We assume programs to be deadlock-free.

is barrier node:
reset T_{SHIFT} and update N_{low} and N_{high}
is memory access node:
add the access to LOG
increment T_{SHIFT} according to N_{low}
update N_{low} and N_{high}
flush accesses from the LOG

The simulation of memory access interleavings is done by keeping a LOG, whose length is limited by T_{SHIFT} and the branching encountered during the simulation. When we flush an access from LOG, we increment T_{WCET} by T_{stall}. In computing T_{stall}, we pessimistically consider the interleavings that can happen within the LOG. Thus, instead of considering whole programs (see, e.g., [5]), we consider interleavings within a small window.

In actual hardware, warps can execute a kernel without synchronization. Our interleaving mechanisms can handle only modest convoying shifts. We can accurately simulate some loop types. Most importantly, these include the loops for which the loop branch is non-divergent.

4 The mini-SIMT machine

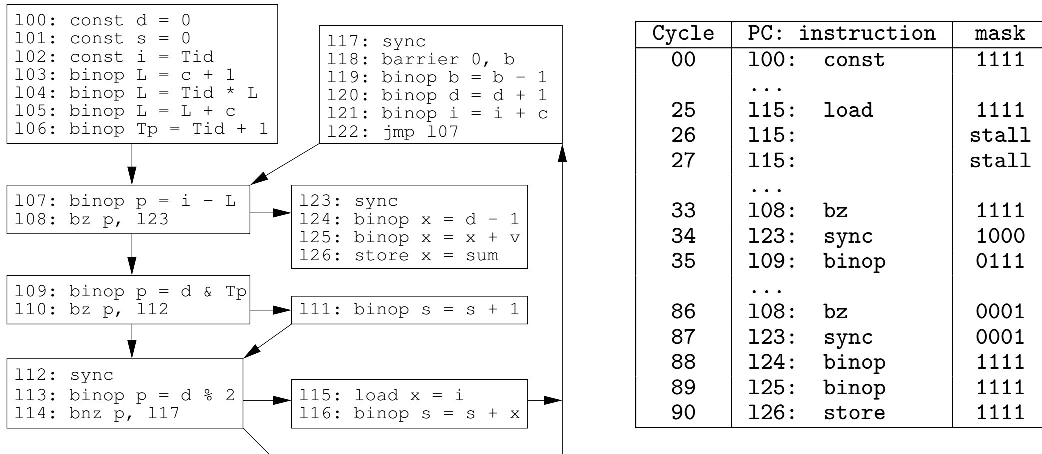
We will use a simple machine language (inspired by [3]):

```
Labels (L)                        ::=   l ∈ N
Variables (V)                     ::=   Tid ∪ {v1,v2,...}
Instructions                      ::=
- (jump if zero/not zero)         |     bz/bnz v,l
- (unconditional jump)            |     jump l
- (store into shared memory)      |     store v_x = v
- (load from shared memory)       |     load v = v_x
- (arithmetic operation)          |     binop v_1 = v_2 op v_3
- (immediate copy)                |     const v = n
- (re-convergence)                |     sync
- (synchronization barrier)       |     barrier v_1,v_2
```

A mini-SIMT program executes kernels with SIMT execution model with multiple cores and uses round-robin scheduling to schedule ready warps. For each warp, the machine keeps a synchronization stack holding frames $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$, where l_{id} is the conditional branch that caused the divergence, Θ_{done} is the set of cores that have reached the synchronization point, l_{next} is the instruction where the set of cores Θ_{todo} will resume execution.

The machine pushes frames at branches (`bz` or `bnz`) and pops them at reconvergence points, but uses a separate reconvergence instruction `sync` to mark the reconvergence points (instead of pushing reconvergence PC onto a stack). Only `load` can cause memory stalls. The `barrier` instruction unschedules the warps that have reached the barrier until v_2 threads are the same barrier (identified by v_1).

The mini-SIMT machine code for our example kernel and its control flow graph is given in Figure 2, whose right side lists excerpts of the kernel executed with 4 threads in a single warp, instruction execution time of 1 cycle, and memory latency of 10 cycles



Cycle	PC: instruction	mask
00	l00: const	1111
	...	
25	l15: load	1111
26	l15:	stall
27	l15:	stall
	...	
33	l08: bz	1111
34	l23: sync	1000
35	l09: binop	0111
	...	
86	l08: bz	0001
87	l23: sync	0001
88	l24: binop	1111
89	l25: binop	1111
90	l26: store	1111

Figure 2 The control flow graph of our translated example program (left) and parts its execution with 4 threads (right), 1 = active thread, 0 = passive thread. Note the pushes and pops by `sync`.

5 An example of analysis

To clarify our method, we consider the program in Listing 1 to be executed on a mini-SIMT machine with 16 threads in 4 warps. In the following, we will first do divergence analysis for the kernel, then we will construct an abstract warp that describes the kernel, and finally, do abstract CTA simulation for it.

5.1 Static divergence analysis

We use GSA in the divergence analysis. Consider the branch at `l08`. GSA places a μ function at `l07` that chains the branch predicate to the definition (`l02: const i = Tid`). As `Tid` is a divergent value, the loop is divergent. Similarly, the branch at `l10` is divergent because its predicate is chained to the definition (`l06: const i = Tid + 1`).

The branch at `l14` is not divergent. GSA places a μ function at `l07` that chains the branch predicate to the definitions (`l00: const d = 0`) and (`l20: const d = d + 1`). Both definitions are non-divergent. Thus, the branch itself and also the memory access controlled by the branch are non-divergent.

5.2 Abstract warp construction

Our abstract warp construction basically follows the CFG given in Figure 2. The number of instructions in the basic blocks give us the related T_{instr} values, because of the execution time of 1 cycle for all instructions. The construction algorithm splits the block of `l15` in two and places a memory access node in between, because the block contains a memory read. Similar splitting happens for the block of `l18`, into which a barrier node is added.

Some nodes can be merged together. Especially, the path (`l09`, `l10`, `l11`, `l12`, `l13`, `l14`, `l15`) is interesting. It is a worst-case path with divergence at `l11`. The corresponding time node gets the values $T_{instr} = 7$ and $T_{shift} = 1$. The resulting abstract warp is in Figure 3.

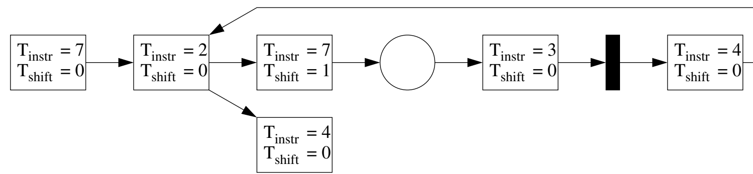


Figure 3 Abstract warp of the example kernel, where boxes are time nodes, the circle is a memory access node, and the bar is a barrier node.

5.3 Abstract CTA simulation

The abstract CTA simulation begins from the leftmost node in Figure 3. As the warp width is 4, we have 4 warps. Thus, the first node adds $7 * 4 = 28$ cycles to T_{WCET} and sets $T_{shift} = 0$. The simulation processes similarly for the next time node.

After that we encounter a divergent branch. Because of the static loop branching predicate we are able to resolve that the iteration counts for the warps are (4, 8, 12, 16). Thus, we proceed with $N_{low} = N_{high} = 4$.

$T_{SHIFT} = 1$ when we encounter the memory access node. The memory access will set $T_{SHIFT} = 3$. Such increasing shift would cause problems later in the simulation, but the barrier node will reset T_{SHIFT} to 0 and flush the read from LOG by increasing T_{WCET} by $10 - N_{low} = 6$ stall cycles.

After four iterations, one warp will diverge from the loop and the simulation continues in the loop with $N_{low} = N_{high} = 3$. After exiting the loop, the simulation continues with $N_{low} = 1$ and $N_{high} = 4$ for the remaining block yielding a final estimate $T_{WCET} = 804$. By using a cycle accurate simulator we obtain 688 as the true execution time.

6 Related work

The timing analysis of programs running on single core machines is rather well known. A survey of the WCET methods applicable for such purposes can be found in [8]. The common methods can be roughly divided into measurement-based methods and methods based on static program analysis. The static program analysis methods typically divide the WCET analysis problem into three sub-problems: flow analysis, processor behavior analysis, and WCET estimate calculation. Some methods have applied model checking, e.g., [4].

Recently, there has been a rising interest on WCET analysis targeting multicore platforms. For example, Gustavsson et. al. [5] present a timing analysis of multithreaded programs on a multicore computer. Their approach applies abstract execution to rather unrestricted programming model. Chattopadhyay et. al. [2] present a unified framework for WCET analysis. The framework is to tackle the problems that have arisen, when the classical approaches have been applied to multicore machines. There has been some work addressing WCET analysis of GPU computing, such as [1].

7 Conclusion

In this paper, we present static timing analysis of GPU programs based on a method that we call abstract CTA simulation. Abstract CTA simulation is based on static analysis of thread divergence in warps and their abstract scheduling.

Our method has obvious limitations. The static divergence analysis can give false positives that lead to over-estimation of execution time. Further, our handling of loops is simplistic.

For divergent loops it can give pessimistic timing, especially when there is complex branching in a kernel.

However, according to our own experience in GPGPU programming, typical kernels are simple in their structure. Many kernels do computation in a map-reduce style, where the mapping phase is essentially non-divergent and the reduction phase is divergent. Often, the actual occupancy is high for the mapping phase and low for the reduction phase. Our approach fits the analysis of such kernels.

Despite its short comings, our methods is very scalable. It can be used to analyze the WCET of very large numbers of parallel threads. This is caused by the fact that the abstract CTA simulation captures efficiently the timing of parallel threads. Abstract CTA simulation spends time in resolving iterations, but typical GPU kernels are short as they rely on massive parallelism.

Currently, our method lacks formal proof of correctness. Further, its applicability is limited by the fact that it has not been integrated with a traditional WCET estimation of the CPU side. Thus only kernels can be analyzed instead of full programs. We see these aspects as the most important topics for further research.

References

1. A. Betts and A. F. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECTRS)*, pages 193–202, 2012.
2. S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s), April 2014.
3. B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Jr. Meira. Divergence Analysis and Optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 320–329, 2011.
4. A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 114–124, 2010.
5. A. Gustavsson, J. Gustafsson, and B. Lisper. Timing Analysis of Parallel Software Using Abstract Execution. In *Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 59–77, 2014.
6. Khronos. OpenCL documentation. `http://www.khronos.org/opencl/`.
7. NVIDIA. CUDA documentation. `http://nvidia.com/`.
8. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, April 2008.

A Time-Predictable Memory Network-on-Chip

Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø

Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk, davidchong99@gmail.com, wopu@dtu.dk, jspa@dtu.dk

Abstract

To derive safe bounds on worst-case execution times (WCETs), all components of a computer system need to be time-predictable: the processor pipeline, the caches, the memory controller, and memory arbitration on a multicore processor. This paper presents a solution for time-predictable memory arbitration and access for chip-multiprocessors. The memory network-on-chip is organized as a tree with time-division multiplexing (TDM) of accesses to the shared memory. The TDM based arbitration completely decouples processor cores and allows WCET analysis of the memory accesses on individual cores without considering the tasks on the other cores. Furthermore, we perform local, distributed arbitration according to the global TDM schedule. This solution avoids a central arbiter and scales to a large number of processors.

1998 ACM Subject Classification B.4.3 Interconnections (Subsystems)

Keywords and phrases Real-Time Systems, Time-predictable Computer Architecture, Network-on-Chip, Memory Arbitration

Digital Object Identifier 10.4230/OASIcs.WCET.2014.53

1 Introduction

The trend in processor design is to increase performance by including more and more processing cores in a single chip. This has been accompanied by a shift from bus-based interconnects to some form of packet switched networks-on-chip (NoC), because chip-wide single-cycle communication has become infeasible. Typically the on-chip processors share an external memory for large shared data structures and for program code. A dedicated NoC is often used for this communication. The NoC and the shared memory are shared resources and in general purpose processors they are a source of timing interferences between tasks executing on different processor cores.

To enable static worst-case execution time (WCET) analysis of applications executing on a multicore processor, both the individual processor cores and the shared memory system (including the NoC) need to be time-predictable [19]. This paper presents the design for timing predictability of the memory interconnect for a chip-multiprocessor. The presented memory system offers time-composability where the execution times of different tasks executing on different processor cores are independent of each other.

Figure 1 shows the multicore platform, as it is developed in the T-CREST project. Several processor cores, the Patmos processors [22], are connected to two NoCs: (1) a core NoC for message passing between processor-local scratchpad memories [7, 21, 24], and (2) a memory NoC – the focus of this paper – that connects all processor cores to the shared, external memory via the memory controller.

The main idea of the presented design is to use TDM scheduling from end to end, such that read or write transactions towards the shared memory are transmitted from the

© Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 53–62

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

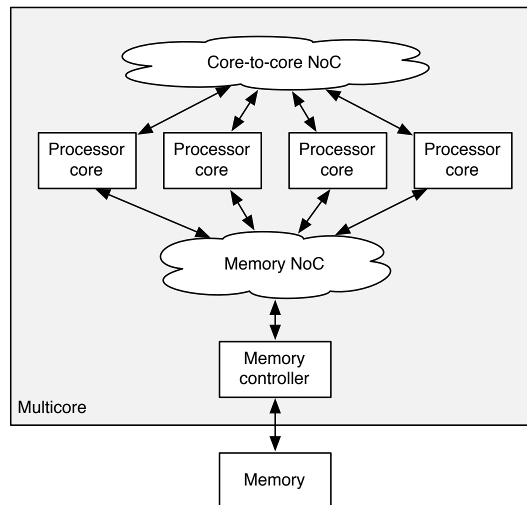


Figure 1 Multicore architecture with several processor cores connected to two NoCs: one for core-to-core message passing and one for access to the shared, external memory.

initiating processor core to the memory without any dynamic arbitration or buffering. Only the processor-local memories (caches and/or scratchpad-memories) buffer any data. By injecting transactions according to a global schedule, they can be propagated in a pipelined fashion even without flow control. The TDM slots and the TDM schedule are defined by the sequence of equally sized read or write transactions towards the memory.

Compared to implementations that use source rate control and dynamic arbitration, the use of TDM results in both a simple hardware implementation and a straightforward WCET analysis. Furthermore, executing this global TDM schedule distributed at the processor cores results in distributed arbitration that scales well with increased number of processor cores. Compared to other TDM memory arbiters we consider pipelining in the design and account for the pipeline delays in the timing parameters. This is the contribution of the paper.

The paper is organized in 6 sections: Section 2 presents related work. Section 3 presents the design of the memory NoC and Section 4 the resulting timing of memory transactions and TDM slotting. Section 5 presents the implementation and evaluation of the memory NoC in an FPGA. Section 6 concludes.

2 Related Work

The design of time-predictable multicore systems is attracting increasing interest. Cullman et al. discusses some high-level design guidelines [3]. To simplify WCET analysis (or even make it feasible) the architecture shall be *timing compositional*. That means that the architecture has no timing anomalies or unbounded timing effects [12]. The Patmos processor, used in the proposed time-predictable multicore, fulfills those properties. For multicore systems the authors of [3] argue for bounded access delays on shared resources. This is in our opinion best fulfilled by a TDM based arbitration scheme, as presented in this paper.

From a structural point of view, communication between processor cores and external memory is different from inter-core communication. While the former follows a many-to-one communication pattern, the latter requires many-to-many communication. Consequently, approaches to make many-to-many communication predictable [5, 13, 24] are not directly comparable to the work presented in this paper.

The proposed memory NoC is similar to "mesh-of-trees" NoCs [2, 17] with a single tree. These NoCs perform 2:1 arbitration in the nodes of the trees such that a request that is blocked by a request on the other input will win arbitration in the next cycle. Therefore, arbitration follows a distributed round-robin scheme.

Different arbitration schemes for a time-predictable memory access of a processor and a video controller are evaluated in [15]. The work has been extended to build a time-predictable multicore version of the Java processor JOP [16]. A TDM based memory arbitration is used and the WCET of individual bytecodes of the Java processor take the possible positions within the TDM schedule into account. Therefore, some latency introduced by TDM arbitration can be hidden. In contrast to our distributed TDM memory arbiter, the JOP TDM arbiter was designed to include the read response within the TDM slot. This design limits the possibility to pipeline the arbiter.

The initial memory connection in the T-CREST project is the so-called Bluetree [6]. Bluetree is a tree of 2:1 multiplexers where the default behavior is that one of the inputs has priority over the other. To avoid starvation of the lower priority input, a counter controls the maximum number of priority messages when a low priority message is pending. This design is optimized to deliver good average-case performance and guarantee worst-case responses. Compared to the Bluetree, our design is not work conserving, but has a shorter worst-case latency guarantee, considering all other parameters the same.

An approach close to our work is presented in [18]. The proposed multicore system is also intended for tasks according to the simple task model [9]. The local cache loading for the processor cores is performed from a shared main memory. Similar to our approach, a TDM based memory arbitration is used. The memory wheel of the PRET architecture [11] is also a form of TDM arbitration. PRET's memory wheel arbitrates between the six hardware threads of processor rather than between different processor cores. Therefore, scalability is not a major concern.

Gomony at al. argue for coupling the NoC TDM slotting with the TDM arbitration within a memory controller [4]. This is in line with our argument. However, compared to their design we completely avoid buffering in the NoC and in the memory controller.

Schranzhofer et al. [23] present a timing analysis for TDM arbitration. They model the case where memory accesses are allowed only at the beginning and end of a task, the case where accesses are allowed at any time, and a hybrid of these two cases. Kelter et al. [8] also present a timing analysis for TDM arbitration. They compare several variants that trade off precision and analysis effort. They find that their approach can lead to significantly lower WCETs for timing-composable systems than for non-composable systems.

Most TDM based designs consider equal slots for all processor cores. However, it is easy to envision a schedule where some cores get more than one slot per TDM period. One can optimize this allocation with the WCET of the individual tasks running on the different processor cores. However, it has been shown that this rather coarse grain optimization is not very efficient [25].

3 Memory Network-on-Chip Design

Figure 2 shows the memory NoC design. As this NoC serves a many-to-one communication flow between several processor cores and a single memory controller, it is organized as a tree. Each processor core is connected via a standard interface, the open core protocol (OCP) [14], to the network interface (NI). The NIs are connected by a tree of merge circuits downstream towards the memory interface (MI) and back upstream for the return data. The

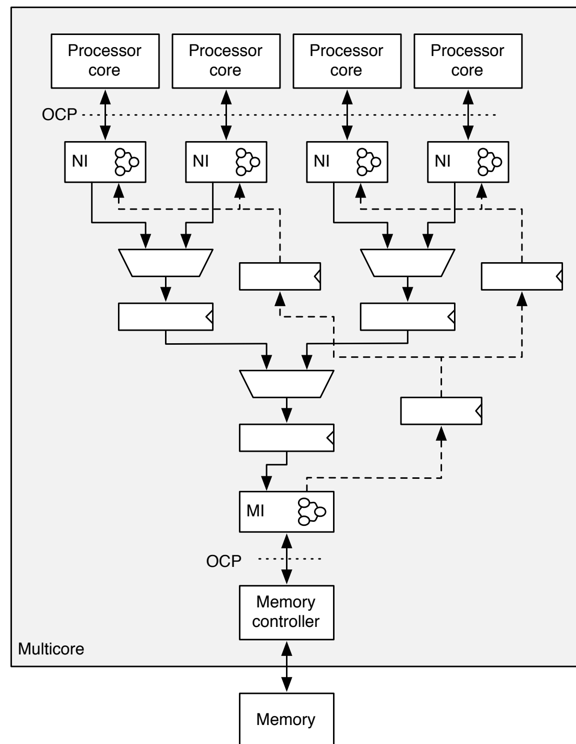


Figure 2 The distributed TDM based memory NoC.

MI is connected to the on-chip memory controller, which itself is connected to the external memory.

The memory NoC supports burst read and write transactions. For single word/byte writes, write enable signals for individual bytes are supported. When the external memory is a DRAM device that needs refresh, a refresh circuit is added to the memory NoC at the same level as a processor core. Therefore, refresh consumes one TDM slot, but has no further influence on the memory access timing.

Each core local NI executes the core relevant part of the global TDM schedule. When the time slot for a core arrives and a memory transaction is pending, the NI acknowledges the transaction to the processor core and the transaction data freely flows down the network tree. No flow control, arbitration, or buffering (except pipeline registers to improve clock frequency) is performed along the downstream path. The memory request arrives at the MI and is translated back to an OCP transaction request to the memory controller. Here OCP handshaking is generated, but the TDM schedule is organized such that it is guaranteed that the memory controller and the memory are ready to accept the transaction.

On a read transaction, the result is returned from the memory to the memory controller and from there back upstream to the processor cores. The returning to the processor cores can be a simple broadcast to all processors, which itself can be pipelined. Alternatively, it can be organized as a broadcast tree as shown in Figure 2. Due to the pipelining, several read requests might be on the fly in the memory NoC, memory controller, and memory. Therefore, a processor might see a read return from a former read request by a different processor after sending the read request.

To identify the correct return data there are two possibilities: (1) either tag the memory transaction with the core number or (2) use time to distinguish between the early and false

Table 1 Timing parameters for the memory interface, the memory controller, and the memory NoC.

Parameter	Meaning
t_b	Burst transfer length
t_{r2b}	Read command to read burst delay
t_{b2e}	Write burst to command end delay
$t_{rd} = t_{r2b} + t_b$	Read transaction timing (at the memory interface)
$t_{wr} = t_b + t_{b2e}$	Write transaction timing (at the memory interface)
t_{ctrlrd}, t_{ctrlwr}	Non-pipelinable time delays in the memory controller
$t_{slot} = \max(t_{rd} + t_{ctrlrd}, t_{wr} + t_{ctrlwr})$	Minimum slot length
N	Number of processor cores
$T = N \times t_{slot}$	TDM period (with equal memory bandwidth)
L_{down}	Memory NoC and controller downstream latency
L_{up}	Memory NoC and controller upstream latency
$t_{wcrd} = T - 1 + L_{down} + t_{slot} + L_{up}$	Worst-case read transaction time
$t_{wcwr} = T - 1 + t_{slot}$	Worst-case write transaction time
$t_{bcrd} = L_{down} + t_{slot} + L_{up}$	Best-case read transaction time
$t_{bcwr} = t_{slot}$	Best-case write transaction time

and the correct read responses. With a tagged transaction and pipelining in the memory controller and memory, the memory controller needs to organize a short queue of the tags of outstanding read requests. As a positive side effect, this information can be used to send the return information only to the originally requesting processor core along the upstream tree, saving power in the other paths. Using time to distinguish between read returns is possible with the static TDM schedule and simple to implement in the local NI. The NI knows the latency for the read request and simply ignores any receiving data until this latency has passed.

For a write transaction there are two options for the handshake between the processor core and the memory NoC: (1) just post the write data and generate the write acknowledgement locally or (2) wait for the write and the write acknowledgment from the memory controller. Just posting the write lets the processor pipeline continue to execute code and hides the memory NoC latency. As the memory NoC does not reorder any memory transactions there is no issue with memory consistency. If the memory and/or controller supports error return codes (e.g., on a parity error) waiting for the response enables error signaling with an exception in the processor core.

The design is configurable with 4 parameters: (1) the level of downstream pipelining, (2) the maximum time for a memory transaction (i.e, the TDM slot length), (3) the additional latency within the memory controller, and (4) the level of upstream pipelining. The pipeline levels influence downstream and upstream latencies.

4 Pipelined Access Timing and Slot Length

We are mainly interested in the worst-case access time for a memory transaction (cache miss). For some scenarios, it might also be interesting what the best-case access time is. For WCET analysis, where we look at a sequence of memory accesses with operations in between, we are interested in how much of the access latency can be overlapped with execution and therefore hidden.

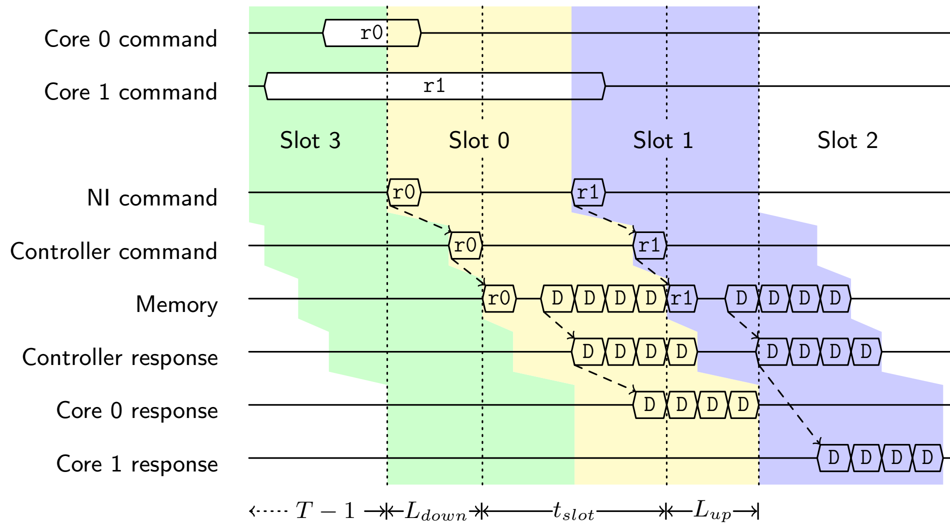


Figure 3 Access latency, TDM slot length, and slot shifting due to pipelining.

Table 1 shows all timing parameters, which are in clock cycles. Timing parameters with a lower case t in the name denote non-pipeline delays and parameters with an upper case L delays that can be pipelined. E.g., the bursting of data via the pins of the memory chips cannot be pipelined, but the burst of data traveling in the memory NoC can be pipelined.

Memory read and write transactions at the memory chip take t_b clock cycles for the burst transfer of the data and some additional latency for a read command to process (t_{r2b}) and a write command to finish (t_{b2e}). Due to some inefficiency in the memory controller, additional delays on read or write transaction (t_{ctrlrd} and t_{ctrlwr}) may be introduced that cannot be hidden by pipelining. The combination of those memory and controller delays determines the minimum TDM slot length t_{slot}.

The slot length t_{slot} and the number of processing cores determine the TDM period T.[1] The memory NoC (and the memory controller) introduce additional latency that needs to be added for the worst-case access time (t_{wcrd} and t_{wcwr}). However, even with a low number of processing cores the TDM period T is the main contributor to the access latency.

Figure 3 shows two read transactions by cores 0 and 1 in a configuration with four cores. The accesses of the cores are delayed until the respective slot arrives; this delay may be up to $T-1$ cycles. From the core's network interface to the memory, a latency of L_{down} cycles is added. Accessing the memory requires t_{slot} cycles. Transmitting the data back to the core adds L_{up} more cycles. The sum of these times determines the access latency observed at a core. The minimum TDM slot length depends however only on the maximum time the memory requires to serve a transaction.

Figure 3 shows that pipelining shifts the TDM slots in time along the path from the processor cores to the memory and back. Therefore, a core's slot starts at different times in different parts of that path. When the timing of the (SDRAM) memory controller is known (e.g., in [10]), the read and write commands can be sent from the client in the exact right point in time so they travel trough the network, arrive at the memory controller, and then arrive at the SDRAM bus without any flow control or buffering (except pipeline registers).

[1] Assuming equal bandwidth for all processing cores. Optimizing the bandwidth of the individual cores for the WCET has not been very beneficial [25].

For WCET analysis the timings for individual cache misses are: t_{wcrd} for a cache line fill and t_{wcwr} for a cache line write back. Compared to a round-robin arbitration, the constant time between two possible accesses can be used to tighten WCET bounds for a sequence of accesses. The time spent executing instructions that do not access memory (or are guaranteed hits) between two accesses can be subtracted from the TDM period T.

5 Implementation and Evaluation

We have implemented three different memory arbiters: (1) a round-robin arbiter, (2) a centralized TDM arbiter, and (3) the distributed TDM arbiter as described in Section 3. We used the high-level hardware description language Chisel [1], which allows the generation of Verilog code for logic synthesis. To evaluate the arbiter rather than the processor cores, we connected the arbiters to test driver cores (4, 8,. . . , 128), together with a PLL, and a memory controller. The design was synthesized and constrained to run at 200 MHz with Altera Quartus II. As target device we chose the Cyclone IV FPGA used on Altera's DE-115 development board (part number EP4CE115F29C7N).

All arbiters use the OCP communication interface used in T-CREST project. For a multicore with N cores, it has N OCP slave interfaces that receive data from the processor cores and one OCP master interface that sends data to the memory controller. To make the arbiter reusable, it is configurable with five parameters: number of cores, address width, data width, burst length, and controller delay. Additional parameters like slot length and period are derived from these parameters.

For a multicore with N cores, each core is given an *id*, $0, 1, \dots N-1$. Based on the number of cores, a specific time slot is defined for each core in the TDM based arbiters. The slot length is defined as $t_{slot} = \max(t_{rd} + t_{ctrlrd}, t_{wr} + t_{ctrlwr})$, which is the amount of clock cycles for a read or write burst. The slot length and the number of cores is configurable. In a complete memory access period, $T = N \times t_{slot}$, each processing core has a chance for either a read or write burst. At each core a counter is used to generate the according single cycle enable signal.

For the centralized TDM arbiter, a finite state machine (FSM) controls the local TDM arbitration. When the *enable* signal is asserted, the FSM will transfer the memory access request of the corresponding node into the memory tree. The FSM has only three states: *idle*, *read*, and *write*.

For the distributed TDM arbiter, the arbitration is divided into N local arbiters. Each local arbiter has a counter for the TDM slot counting. The outputs of all local arbiters are fed to a tree of OR gates and into pipeline registers before reaching the memory controller. On the return path from the memory controller, this data is broadcasted to all the local arbiters. Two pipeline registers on the downstream path and one on the upstream path are enough to keep the critical path short enough for a 200 MHz operating frequency.[2]

Figure 4 shows the maximum frequency (Fmax) of each arbiter. The centralized arbiters (round-robin and central TDM) lead to slower clock frequency when increasing the number of processing cores. The simple round-robin arbiter can be clocked faster than the central TDM arbiter. In contrast, the maximum frequency of the distributed TDM arbiter remains relatively close to the input frequency (200 MHz), even as the number of nodes increases exponentially. The distributed TDM arbiter has the shortest critical path since the datapaths

[2] A RISC style processor core can be clocked at about 100 MHz in this FPGA family.

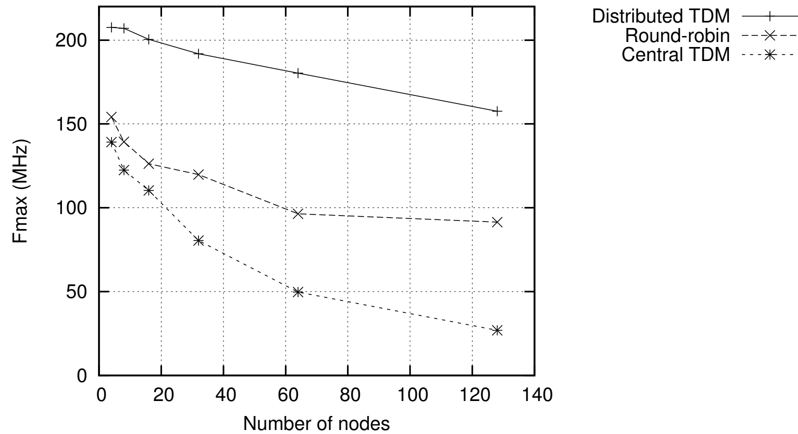


Figure 4 Maximum frequency (Fmax) for different arbiters and number of processor cores.

Table 2 Logic cell counts for different arbiters and different number of processor cores.

Number of processor cores	4	8	16	32	64	128
Round-robin	139	324	614	1267	2549	5114
Centralized TDM	235	446	969	1943	3893	7764
Distributed TDM	470	980	1894	3827	7575	10277

from each node to the memory, through the arbiter, are independent of each other and broken up with pipeline registers.

Table 2 shows the resource consumption of each arbiter in terms of logic cell (LC) count on the FPGA. To set the number in relation, a RISC style processor pipeline consumes about 2000–5000 LCs. The centralized TDM arbiter requires more logic cells than the round-robin arbiter.

The distributed TDM arbiter replicates some logic (e.g., the counters for the TDM slots) at the client side. Therefore, it consumes more resource, but also allows the highest clock frequency. Each client side component consumes about 30 LCs. However, the memory tree, made up of OR gates, consumes a large number of LCs. For every 4 cores, a 4-input look-up table (LUT) is needed for every bit of the OCP signals. As a rough estimate, about 55 logic cells are needed for the OR-gate tree per node. With a back-of-an-envelope calculation the LCs needed for the 128-core multicore can be estimated as: 30 LC per core * 128 + 55 LC per OR-gate tree * 128 = 10880. This confirms the synthesized results shown in Table 2.

Compared to centralized TDM arbitration, as used in [11] and [16], our pipelined design with the distributed arbitration at the individual nodes scales better with more nodes. The additional cost is moderate. The cost per node is in the range of 120 LCs, where a RISC style processor node itself consumes between 2000 and 5000 LCs.

6 Conclusion

Computer systems for real-time systems need to be time-predictable to allow static WCET analysis. For multicore processors with shared memory the access to this shared memory needs to be time-predictable as well. A time-division multiplexing (TDM) arbiter is time-predictable. It allows calculating the WCET of a task executing on one processor core

independent from other tasks executing on other cores. This paper presented a TDM memory network-on-chip with distributed arbitration and pipelining to achieve a high throughput through the interconnect. The additional latency through pipelining does not influence the slot length of the TDM schedule. The TDM schedule and the knowledge of the pipelined interconnect is the input for the WCET analysis of memory accesses for a multicore processor.

Source Access

The source of the described memory NoCs is open source under the simplified BSD licenses and available at GitHub within the Patmos project: `https://github.com/t-crest/patmos`. Details on the build process can be found in the Patmos reference handbook [20].

Acknowledgment. This work is part of the project "Hard Real-Time Embedded Multiprocessor Platform – RTEMP" and received partial funding from the Danish Research Council for Technology and Production Sciences under contract no. 12-127600. This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

References

1 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

2 Aydin O Balkan, Gang Qu, and Uzi Vishkin. Mesh-of-trees and alternative interconnection networks for single-chip parallelism. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(10):1419–1432, 2009.

3 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

4 Manil Dev Gomony, Benny Akesson, and Kees Goossens. Coupling tdm noc and dram controller for cost and performance optimization of real-time systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

5 Manel Djemal, François Pêcheux, Dumitru Potop-Butucaru, Robert De Simone, Franck Wajsburt, and Zhen Zhang. Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–8, Oct 2012.

6 Jamie Garside and Neil C Audsley. Investigating shared memory tree prefetching within multimedia noc architectures. In *Memory Architecture and Organisation Workshop*, 2013.

7 Evangelia Kasapaki and Jens Sparsø. Argo: A Time-Elastic Time-Division-Multiplexed NoC using Asynchronous Routers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE Computer Society Press, 2014.

8 Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014.

9 Herman Kopetz. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.

10 Edgar Lakis and Martin Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

11 Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

12 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

13 Jörg Mische, Stefan Metzlaff, and Theo Ungerer. Distributed memory on chip–bringing together low power and real-time. In *Proceedings of the Workshop on Reconciling Performance and Predictability (RePP)*, Grenoble, France, 2014.

14 OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

15 Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 317–322, Amsterdam, Netherlands, August 2007. IEEE.

16 Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

17 Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

18 Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*, pages 49–60, Dec. 2007.

19 Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

20 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. *Patmos Reference Handbook*. Technical University of Denmark, 2014.

21 Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE.

22 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

23 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In Marco Caccamo, editor, *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, pages 215–224. IEEE Computer Society, 2010.

24 Jens Sparsø, Evangelia Kasapaki, and Martin Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1044–1047, San Jose, CA, USA, 2013. EDA Consortium.

25 Jack Whitham and Martin Schoeberl. The limits of TDMA based memory access scheduling. Technical Report YCS-2011-470, University of York, 2011.

The Challenge of Time-Predictability in Modern Many-Core Architectures*

Vincent Nélis[1], Patrick Meumeu Yomsi[1], Luís Miguel Pinho[1], José Carlos Fonseca[1], Marko Bertogna[2], Eduardo Quiñones[3], Roberto Vargas[3], and Andrea Marongiu[4]

1 **CISTER/INESC-TEC Research Center, Porto, Portugal**
{nelis, pamyo, lmp, jcnfo}@isep.ipp.pt
2 **University of Modena, Italy**
marko.bertogna@unimore.it
3 **Barcelona Supercomputing Center, Spain**
{eduardo.quinones, rvargas}@bsc.es
4 **IIS – ETH Zürich, Switzerland**
a.marongiu@iis.ee.ethz.ch

Abstract

The recent technological advancements and market trends are causing an interesting phenomenon towards the convergence of High-Performance Computing (HPC) and Embedded Computing (EC) domains. Many recent HPC applications require huge amounts of information to be processed within a bounded amount of time while EC systems are increasingly concerned with providing higher performance in real-time. The convergence of these two domains towards systems requiring both high performance and a predictable time-behavior challenges the capabilities of current hardware architectures. Fortunately, the advent of next-generation many-core embedded platforms has the chance of intercepting this converging need for predictability and high-performance, allowing HPC and EC applications to be executed on efficient and powerful heterogeneous architectures integrating general-purpose processors with many-core computing fabrics. However, addressing this mixed set of requirements is not without its own challenges and it is now of paramount importance to develop new techniques to exploit the massively parallel computation capabilities of many-core platforms in a predictable way.

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

Keywords and phrases Time-Predictability, Many-Cores, Multi-Cores, Timing Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2014.63

1 Current Trends in Application Requirements

Nowadays, computing systems are subject to a wide continuum of requirements, spanning from high-performance computing (HPC) systems to real-time embedded computing (EC) systems. Sitting on one extremity of that spectrum, HPC systems have been for a long time the realm of a specific community within academia and specialized industries; in particular,

* This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-037281 (CISTER), and by the European Union, under the Seventh Framework Programme (FP7/2007-2013), grant agreement n° 611016 (P-SOCRATES), and by EU project TACLe (ICT COST Action IC1202).

© Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, José Carlos Fonseca, Marko Bertogna, Eduardo Quiñones, Roberto Vargas, and Andrea Marongiu;
licensed under Creative Commons License CC-BY

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 63–72

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

those targeting demanding analytic- and simulation-oriented applications that require massive amounts of data to be processed. For HPC system designers, "the faster, the better" is the mantra. On the other side of the spectrum, EC systems have also focused on very specific systems; in particular those with pre-set and specialized functionalities for which timing requirements prevail over performance requirements. Historically, the key objective for designers of EC systems was to design highly predictable systems where the time taken by every computing operation is *upper-bounded and these upper-bounds are known at design time*; Being fast was secondary.

With the new generation of computing platforms and the ever-increasing demand for safer but more complex applications, the conceptual boundary that was pulling HPC and EC systems apart is getting thinner every day. HPC systems require more and more guarantees on the timing behavior of their applications while EC systems face an increasing demand for computational performance. As a result, these HPC and EC systems that used to be torn apart by orthogonal requirements are now converging towards a brand new category of systems that share both HPC and EC requirements. This is the case of real-time complex event processing (CEP) systems [6], a new area of computing systems that literally cross the boundaries between the HPC and the EC domains.

In these CEP systems, the data come from multiple event streams and is correlated in order to extract and provide meaningful information within a pre-defined time bound. In cyber-physical systems for instance, ranging from automotive and aircraft to smart grids and traffic management, CEP systems are embedded in a physical environment and their behavior obeys technical rules dictated by this environment. Another example is the banking/financial markets where CEP systems process large amounts of real-time stock information in order to detect time-dependent patterns, automatically triggering operations in a very specific and tight time-frame when some pre-defined patterns occur [12].

The underlying commonality of the systems described above is that they are time-critical (whether business-critical or mission-critical) and with high-performance requirements. In other words, for such systems, the correctness of the result is dependent on both performance and timing requirements, and the failure to meet either of them is critical to the functioning of the system. In this context, it is essential to guarantee the timing predictability of the performed computations, meaning that arguments and analysis are needed to be able to make arguments of correctness, e.g., performing the required computations within well-specified time bounds.

2 Trends in the High-performance and Embedded Computing Domains

Until now, trends in high-performance and embedded computing domains have been running in opposite directions. On the one hand, HPC systems are traditionally designed to make the common case as fast as possible, without concerning themselves for the timing behavior (in terms of execution time) of the not-so-often cases. The techniques developed for HPC are usually based on complex hardware and software structures that make any reliable time bound almost impossible to derive. On the other hand, real-time embedded systems are typically designed to provide energy-efficient and predictable solutions, without heavy performance requirements. Instead of *fast* response times, they aim at having *predictable* response times, in order to guarantee that deadlines are met in all possible execution scenarios. Hence these systems are typically based on simple hardware architectures, using fixed-function hardware accelerators that are strongly coupled with the application domain.

This section presents the evolution of both the HPC and the EC computing domains from a hardware and software point of view.

2.1 Hardware Trends

Owing to the immense computational capabilities needed to satisfy the performance requirements of HPC systems and because the resulting exponential increments of power requirements exceeded the technological limits of classic single-core architectures (typically referred to as the power-wall), multi-core processors have entered both computing markets in the last years [11]. The leading hardware manufacturers are now offering an increasing number of computing platforms that integrate multiple cores within a chip, which contributes to an unprecedented phenomenon sometimes referred to as the multi-core revolution.

Multi-core processors are much more energy-efficient and have a better performance-per-cost ratio than their single-core counterpart as they improve the application performance by exploiting thread-level parallelism (TLP). Applications are split into multiple tasks that run in parallel on different cores, which has for consequence to spread into the multi-core world an important challenge that was already faced by HPC designers at multi-processor system level: the parallelization. In the HPC domain, many-core platforms are seen as a highly scalable multi-core architecture that overcomes the limits of traditional multi-cores (such as the contention for memory bus for example) and considerably increases the degree of parallelization of the tasks that can be exploited.

In the EC domain, the necessity of developing more flexible and powerful systems have pushed the embedded market in the same direction. For instance, the mobile phone market evolved from selling cellphones with a limited number of well-defined functions to selling smart-phones and tablets with an unlimited access to a virtual store full of user-made applications. As newest applications are more and more greedy in term of performance, multi-core architecture have been increasingly considered as the solution to cope with the performance and cost requirements [3], because they allow multiple application services to be scheduled on the same processor, which maximizes the hardware utilization while reducing its cost, size, weight and power requirements. Unfortunately, most of multi-core architectures have been designed to provide increased performance rather than towards offering time-predictability to the application system and broadly speaking, these platforms failed to provide an appropriate execution environment to time-critical embedded applications. This is why those applications are still executed on simple architectures that are able to guarantee a predictable execution pattern while avoiding timing anomalies [7], which makes real-time embedded platforms still relying on either single-core or simple multi-core CPUs, integrated with fixed-function hardware accelerators into the same chip: the so-called System-on-Chip (SoC).

The needs for time-predictability, energy-efficiency, and flexibility, coming along with Moore's law greedy demand for performance and the advancements in the semiconductor technology, have progressively paved the way for the introduction of many-core systems in both the HPC and EC domains. Examples of many-core architectures include the Tilera Tile CPUs [13] (shipping versions feature 64 cores) in the embedded domain and the Intel MIC [4] and Intel Xeon Phi [5] (featuring 60 cores) in the HPC domain. The introduction of many-core systems has set up an interesting trend wherein both the HPC and the real-time embedded domains converge towards similar objectives and requirements. Figure 1 shows the trend towards the integration of both domains. In this current trend, challenges that were previously specific to each computing domain start to be common to both (including energy-efficiency, parallelisation, compilation, software programming) and are magnified by

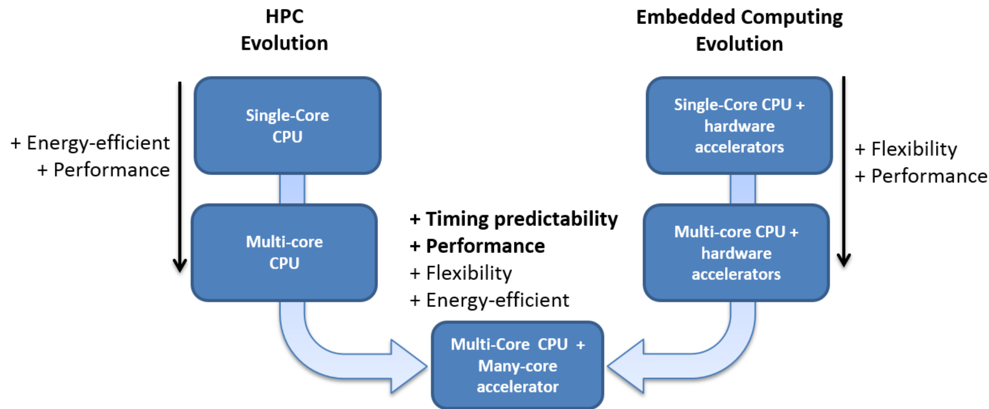


Figure 1 Trend towards the integration of HPC and embedded computing platforms.

the ubiquity of many-cores and heterogeneity across the whole computing spectrum. In that context, cross-fertilization of expertise from both computing domains is mandatory. In our opinion, there is still one fundamental requirement that has not yet been considered: time predictability as a mean to address the time criticality challenge when computation is parallelised to increase the performance. Although some research in the embedded computing domain has started investigating the use of parallel execution models (by using customized hardware designs and manually tuning applications by using specialized software parallel patterns [10]), a real cross-fertilization of expertise between HPC and embedded computing domains is still missing.

2.2 Software Trends

Industries with both high-performance and real-time requirements are eager to benefit from the immense computing capabilities offered by these new many-core embedded designs. However, these industries are also highly unprepared for shifting their earlier system designs to cope with this new technology, mainly because such a shift requires adapting the applications, operating systems, and programming models in order to exploit the capabilities of many-core embedded computing systems. Neither many-core embedded processors have been designed to be used in the HPC domain, nor HPC techniques have been designed to apply embedded technology. Furthermore, real-time methods that determine the timing behavior of an embedded system are not prepared to be directly applied to the HPC domain and many-core platforms, leading to a number of significant challenges. Although customized processor designs could better fit real-time requirements [10], the design of specialized processors for each real-time system domain is not a desired option for obvious financial reasons.

Different parallel programming models and multiprocessor operating systems have been proposed and are increasingly being adopted in today's HPC computing systems. In recent years, the emergence of accelerated heterogeneous architectures such as GPGPUs, have introduced parallel programming models such as OpenCL [9], the currently dominant open standard for parallel programming of heterogeneous systems, or CUDA [8], the dominant proprietary framework of NVIDIA. Unfortunately, they are not easily applicable to systems with real-time requirements since, by nature, many-core architectures are designed to integrate as many functionalities as possible into a single chip and thus they inherently share as many resources as possible amongst the cores, which heavily impacts the ability to provide timing guarantees.

The embedded computing domain world has always seen many application-specific accelerators with custom architectures on which applications are manually tuned to achieve predictable performance. Such types of solutions have a limited flexibility which complicates the development of embedded systems. However, we firmly believe that commercial-off-the-shelf (COTS) components based on many-core architectures are likely to dominate the embedded computing market in the near future. Assuming that embedded systems will evolve in this way, migrating real-time applications to many-core execution models with predictable performance requires a complete redesign of current software architectures. Real-time embedded application developers will therefore either need to adapt their programming practices and operating systems to future many-core components, or they will need to content themselves with stagnating execution speeds and reduced functionalities, relegated to niche markets using obsolete hardware components. This new trend in the manufacturing technology, alongside the industrial need for enhanced computing capabilities and flexible heterogeneous programming solutions of accelerators for predictable parallel computations, bring to the forefront important challenges for which solutions are urgently needed. To that end, we envision the necessity to bring together next-generation many-core accelerators from the embedded computing domain with the programmability of many-core accelerators from the HPC computing domain, supporting this with real-time methodologies to provide time-predictability. Time-predictability is an essential feature to allow system designers to model the timing behavior of the system through timing analysis techniques and then, based on these models, check that all its timing requirements are fulfilled.

3 Background on timing analysis techniques

What is it?

Timing analysis is any structured method or tool applied to the problem of obtaining information about the execution time of a program, a part of a program, or even any kind of computer operation such as a fetching a data in the cache or sending a packet over a network. The fundamental problem that timing analysis techniques have to deal with is the fact that the execution time of an operation is not a fixed constant, but rather varies across a range of possible execution times. Variations in the execution time of an operation occur due to variations in input data, as well as the characteristics and execution history of the software, the processor architecture, and the computer system in which the operation is executed.

What is it needed for?

Timing analysis is needed to assess that all the timing requirements of the system are fulfilled. In the EC domain, most of systems with real-time requirements require a reliable timing analysis to be efficiently designed and verified, in particular when the system is used to control safety critical components in application areas such as vehicles, aircraft, medical equipment, and industrial plants. In these application domains, in order for the whole system to be validated and assessed as safe, it is commonplace that only a subset of tasks has to fulfill strict timing requirements (i.e., they are required to complete their operations within specified time limits). That is, only few components of the entire system are “critical” and in need of precise timing analysis. An accurate timing analysis is consequently not always required as many components may be subject to real-time requirements but are in essence not critical. It is currently the case, for example, for most of modern applications that share HPC and real-time requirements.

Although the high criticality of some applications is beyond doubt, for many functions it is rather a business matter to evaluate whether the costs and consequences of a timing-related failure is worth the cost of the various mechanisms that must be implemented to prevent/handle this failure. In industrial systems, there is a continuum of criticality levels in the set of components of a real-time system. Depending on the criticality of each component an approximate or less accurate analysis might be acceptable. Real-time applications are commonly categorized as safety-critical (or life-critical), mission-critical, and non-critical. A failure or malfunction of a safety-critical application may result in death or serious injury to people, loss or severe damage to equipment or environmental harm, whereas a failure of a mission-critical application may result in a failure of the entire system, but without damaging it nor its embedding environment, and a failure of a non-critical application has no severe consequences. While safety- and mission-critical components must be certified with a very high level of confidence (through extremely accurate and thorough analyses), components that are less or not critical at all need only to maintain a "decent" average throughput and should be proven not to affect the execution behavior of the critical components.

How does timing analysis work?

The worst-case execution time of an operation depends not only on the intrinsic nature of the operation and its inner sub-operations, but also on external events that may occupy or lock a resource that the operation needs to access. For example, the worst-case traversal time of a packet throughout a network-on-chip does not only depend on intrinsic properties like the size of the data sent, the routing algorithm employed, or the capacity of the links between the source and the destination, but also on the current traffic on the network at the time the packet is sent. This means that, in order to provide a safe upper-bound on the execution time of an operation, timing analysis techniques must consider not only the nature of the operation and the characteristics of the executing environment, but also they must identify the worst "context" in which the operation can be performed. Owing to this influence of the context of execution, the body of knowledge developed in academia further sub-categorizes the timing analysis objectives and distinguishes between (i) the worst-case execution time (WCET) analysis and (ii) the interference analysis.

What is WCET analysis?

The *WCET analysis* is the context-independent part of the timing analysis that focuses on deriving a safe upper-bound on the execution time of a program (or any piece of code). It assumes that the analyzed program runs in isolation and without interruption, i.e., there is no other user-tasks running concurrently with the analyzed task, interrupts from the operating system are disabled, and the analyzed task gets immediate access to a resource as soon as it needs to. Under these circumstances, the WCET of a program is defined as the longest execution time that will ever be observed when the program is run on the target hardware. It is the most critical measure for most real-time work. For example, as mentioned earlier, the WCET of tasks is a key component for the higher-level schedulability analysis, but in practice it is also used at a lower level analysis, e.g., to ensure that software interrupts will have sufficiently short reaction times, or to guarantee that operating system calls return to the user application within pre-defined time-bounds.

WCET analysis can be performed in a number of ways using different tools, but the main methodologies employed can be broadly classified in three categories: (1) static analysis techniques, (2) measurement-based analysis techniques, and (3) hybrid techniques. Broadly

speaking, measurement-based techniques are suitable for software that are less time-critical and for which the average-case behavior (or a rough WCET estimate) is more meaningful or relevant than an accurate estimate. For example, systems where the worst-case scenario is extremely unlikely to occur and/or the system can afford to ignore it if it does occur. For highly time-critical software, where every possible execution scenario must be covered and handled, the WCET estimate must be as accurate as possible and static analysis or some type of hybrid method is therefore preferable.

What is interference analysis?

The *interference analysis* is the execution-context aware part of the timing analysis. It focuses on deriving safe upper-bounds on the extra execution time-penalty that the analyzed task may suffer during its run-time because of the interference with other tasks and with the system. It takes into account the context in which each operation is performed and identifies the worst-case interference scenario for the analysis. Typically, interference analysis will supplement the outputs of the WCET analysis by factoring in extra delays due to, for example, sporadic SW/HW interrupts or the interference from other tasks on the shared communication bus, network, or caches. Specifically, for every shared software and hardware components (such as the caches, the main memory, the shared data, etc.) and for each access to these resources that the analyzed task may request, the interference analysis techniques identify the worst initial "state" of the component and the worst-case scenario of interference (from the system and the other tasks) on that component that would induce the largest execution time for the analyzed access. These upper-bounds are then used to adjust the WCET estimates obtained from the WCET analysis techniques. It must be noted that both the WCET analysis techniques and the interference analysis techniques may analyze the same SW/HW resources, but their main focus and objectives are not the same. For example, some WCET analysis tools include cache analysis, during which the tool may substantially tighten the WCET estimate by taking into account that the requested data will not always have to be fetched from the main memory as it may have been loaded already and is thus available in the local cache. In contrast, interference analysis techniques also analyze the cache(s) but relax the assumption that the analyzed task is the only one that can use it, thus allowing tasks to evict cache lines from each other. Relaxing this assumption trivially causes the tasks to experience extra delays during their execution and the WCET estimates must therefore be augmented accordingly.

Interference between tasks and applications are typically reduced by ensuring a certain degree of "isolation" between those tasks and applications. Isolation can come in different flavors: tasks can be isolated in the time domain, the space domain, or both, and it can be symmetric or asymmetric. Isolation between system components also provides other advantages: it is fostered by system designers to avoid fault propagation for example, and when system timeliness is of concern, it helps provide two major features:

- **Time compositionality:** the timing properties of interest at system level can be determined from the timing properties of its constituent components.
- **Time Composability:** the timing properties determined for individual components in isolation should hold after the composition with other components.

Typically, these two properties (and in particular the time-composability property) are obtained by enforcing spatial and temporal isolation between software components at run-time.

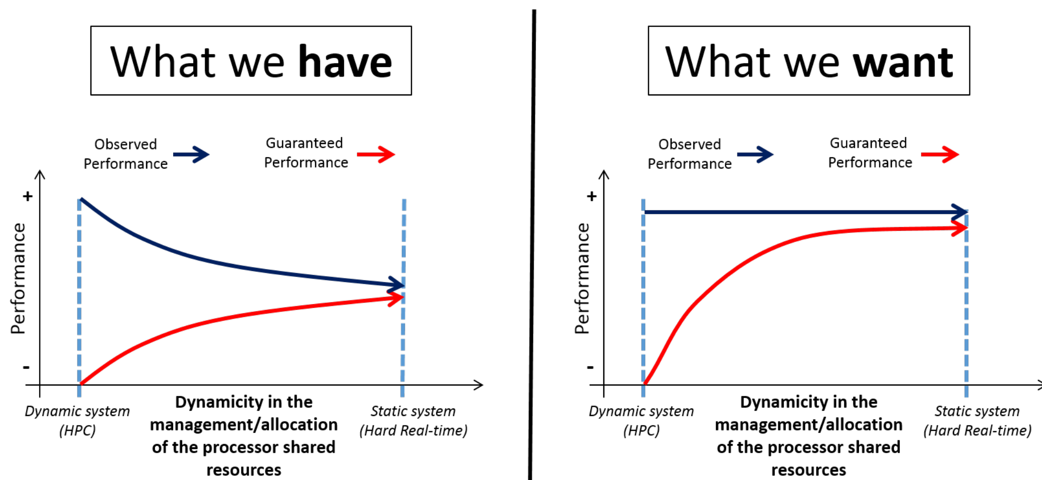


Figure 2 Performance degradation and guarantees improvement.

4 Glance at a few forthcoming challenges in ensuring time-predictability

The challenge of ensuring time-predictability for this new generation of systems with mixed requirements is twofold. On one side, the software solutions used in HPC systems must be adapted to be more predictable while preserving (as much as possible) their efficiency and on the other side, timing analysis techniques used to validate EC systems must be adapted to these new software solutions.

What does it imply to adapt the HPC software solutions?

It mostly implies reducing the dynamicity of all the mechanisms that are responsible for the management of the communication, memory, and computing resources. Instead of taking decisions on-the-fly based on the execution history and/or the current state of the system (as it is done in HPC systems), most of the decisions regarding the allocation of the resources among the tasks should ideally be taken before the run-time to enable a thorough offline analysis of the system timing behavior. Figure 2 (left side) illustrates the expected trends in the (observed) average performance and in the guaranteed performance when the dynamicity of the resource allocation schemes is reduced. Typically, as we shift the decision-taking process from the run-time to the design-time we limit the dynamicity of the system, which has for effect to decrease the observed performance as the system becomes less “flexible” while the guaranteed performance increases as the system becomes more predictable. The challenge here is to obtain high performance and tight guarantees of this high performance as depicted on the right-hand side of Figure 2 or, if this turns out not to be possible, one should at least find an appropriate trade-off between the observed performance and the guaranteed performance.

What are the changes needed at the programming model level?

In a nutshell, programming models need to be extended to provide detailed information about the code including for instance information on the control flow, timing properties, and functional and data dependencies between parts of the code. These annotations of the code

could be used to extract an accurate and complete model of the application where all the dependencies between the functions (or any piece of code) are clearly documented. Together with the control flow information and the estimations of the worst-case execution time of each part of the code, this information could be used by timing analysis techniques to derive safe bounds (exact or probabilistic) on the overall execution time of the application.

To the best of our knowledge, the greatest effort in that direction is provided at Barcelona SuperComputing Center (BSC) where researchers have developed OmpSs, a programming model that integrates features from the StarSs programming model developed also at BSC into a single programming model. In particular, the objective of OmpSs is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs).[1] However, it can also be understood as new directives extending other accelerator based APIs like CUDA or OpenCL. The OmpSs environment is built on top of the Mercurium compiler and the Nanos++ run-time environment. More details about OmpSs and its objective can be found in [2].

Once we have a time-predictable setup, can we apply commercially-available WCET analysis tools?

It is very unlikely that all the existing methods will be applicable to the next-gen applications that share HPC and real-time requirements, especially it is the case for static approaches. Although static approaches have proven to be very efficient for safety-critical embedded systems these next-gen applications are not (yet?) safety-critical even though they present real-time requirements, which means that they are not subject to the hard and fast programming rules that are idiosyncratic to the safety-critical domain. They typically use pointers, dynamic memory allocation, recursive functions, variable-length loops, etc., and sometimes these applications are implemented by third party companies that are not concerned at all by the validation of the overall system, i.e. the code is not annotated with timing-related information that could be helpful for the timing analysis like loop-bounds for instance. Because of the lack of strict programming rules and the lack of information related to timing aspects of the code, static approaches are likely to fail to provide tight upper-bounds on the execution time and we foresee a rising popularity of measurement-based and probabilistic approaches in a near future.

Furthermore, it must be noted that it is unreasonable (if not impossible) to perform an exhaustive testing of these next-gen applications. Besides the fact that the size and the complexity of the software are constantly increasing, forthcoming platforms may chose to continue to increase their performance by borrowing more and more techniques from the HPC domain, including advanced computer architecture features such as caches, pipelines, branch prediction, and out-of-order execution. These features increase the speed of execution on average, but also make the timing behavior much harder to predict by parsing the code, since the variation in execution time between fortuitous and worst cases increases. The problem was already central in single-core platforms but is now further exacerbated in a multi/many-core setting where low-level hardware resources like caches and communication medium are shared by several cores, thereby inducing situations in which several entities contend for accessing the same resource.

[1] Note that this objective has been achieved by now and the latest version of openMP already integrates the research results obtained at BSC in that domain.

And what about interference analysis techniques?

As introduced before, the existing WCET techniques cannot be applied as is and need to be augmented by further analyses to factor in all the extra delays due to contention for the shared resources. Although preliminary results have already been presented in that direction (see [1] for a list of potential sources of interference between tasks), most of the scientific papers on the subject present techniques to estimate the extra delay due the contention for a single shared resource. That is, the authors focus on one and only one source of interference at a time, such as the cache, the network, the memory bus, etc., while the challenge of making these analyses work together has almost never been studied and we firmly believe that interference analysis need to be tackled in a holistic, integrated perspective.

References

1 Dakshina Dasari, Bjorn Andersson, Vincent Nelis, Stefan M. Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Proceedings of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM'11, pages 1068–1075, Washington, DC, USA, 2011. IEEE Computer Society.

2 Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193, 2011-03-01 2011.

3 T. Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. In *IEEE Micro, Special Issue on European Multicore Processing Projects*, volume 30:5, pages 66–75. IEEE Computer Society, aug 2010.

4 Intel Corporation. *Intel Many Integrated Core (MIC) Architecture*, last access Nov 2013. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integratedcore/intel-many-integrated-core-architecture.html`.

5 Intel Corporation. *Intel Xeon Phi*, last access Nov 2013. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

6 David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., 2001.

7 T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

8 NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture, Version 2.0*, 2008.

9 OpenCL. *The open standard for parallel programming of heterogeneous systems*, 2013. `http://www.khronos.org/opencl/`.

10 parMERASA FP7 European Project – grant agreement 287519. *Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability*, 2011–2014. `http://www.parmerasa.eu`.

11 Sutter, Herb. *Welcome to the Jungle.* `http://herbsutter.com/welcome-to-the-jungle/`.

12 R. Tieman. Algo trading: the dog that bit its master. *Financial Times*, March, 2008.

13 Tilera Corporation. *Tile Processor, User Architecture Manual, release 2.4, DOC.NO. UG101*, May 2011.

Scope-Based Method Cache Analysis*

Benedikt Huber[1], Stefan Hepp[2], and Martin Schoeberl[3]

1 **Institute of Computer Engineering**
Vienna University of Technology
`benedikt@vmars.tuwien.ac.at`

2 **Institute of Computer Languages**
Vienna University of Technology
`hepp@complang.tuwien.ac.at`

3 **Department of Applied Mathematics and Computer Science**
Technical University of Denmark
`masca@dtu.dk`

Abstract

The quest for time-predictable systems has led to the exploration of new hardware architectures that simplify analysis and reasoning in the temporal domain, while still providing competitive performance. For the instruction memory, the method cache is a conceptually attractive solution, as it requests memory transfers at well-defined instructions only. In this article, we present a new cache analysis framework that generalizes and improves work on cache persistence analysis. The analysis demonstrates that a global view on the cache behavior permits the precise analyses of caches which are hard to analyze by inspecting cache state locally.

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

Keywords and phrases Real-Time Systems, Cache Analysis, Time-predictable Computer Architecture

Digital Object Identifier 10.4230/OASIcs.WCET.2014.73

1 Introduction

In this paper, we are concerned with instruction cache architectures for real-time systems, and in particular with their analysis. For time-predictable architectures, we expect that it is possible to compute precise worst-case execution time (WCET) bounds for a sequence of instructions by only considering the number of cache misses, instead of the actual cache state at each instruction. Time-compositional architectures, such as Patmos [17], that enjoy this property are not just simpler to analyze, they also facilitate a larger class of timing analysis techniques and reduce the problem of interfering components, such as data and instruction caches. For non time-compositional architectures, global analyses like ours are still applicable, but need to be combined with local classification using prediction graphs [1].

The method cache was devised as an alternative to set-associative instruction caches for time-predictable architectures [16]. In contrast to traditional instruction caches, it does not manage individual cache lines, but holds entire blocks of code of variable size, e.g., an entire function. Conceptually, the advantage of the method cache is that only few, compiler-controlled instructions may cause a cache miss. This can be exploited to better control instruction-memory related latencies, to avoid interferences between instruction and

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

© Benedikt Huber, Stefan Hepp, and Martin Schoeberl;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 73–82

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

data cache competing for main memory access, and to optimize instruction cache usage. The performance of the method cache and the quality of worst-case guarantees for this cache, however, strongly depend on the compiler and the availability of a precise cache analysis.

In this article, we show that with a global view on the cache behavior, it is possible to effectively analyze instruction caches using the first-in-first-out (FIFO) replacement policy. We therefore invalidate previous assumptions that the method cache, which uses a FIFO replacement strategy, is intrinsically hard to analyze.

The key contribution of this article is a novel cache analysis that is applicable to various instruction cache architectures. To the best of our knowledge, this is the first scalable and precise analysis technique for method caches with variable code block size and FIFO replacement. Our cache analysis is also applicable to standard set-associative caches using either FIFO or the least-recently-used (LRU) replacement strategy, and is thus applicable to a wide range of target platforms.

The paper is organized as follows: Section 2 provides background information on the method cache and its variants. Section 3 presents our scope-based cache analysis, which is evaluated in Section 4. Section 5 discusses related work, Section 6 concludes the paper.

2 Method Cache

In contrast to set-associative caches, the method cache stores whole blocks of code of variable size [16, 2], Cache entries are allocated and evicted only at certain instructions such as call and return instructions, and are in general considerably larger than a cache line of a set-associative cache.

This design brings several advantages. It requires less memory for cache tags than a set-associative cache, there are no interferences with the data cache to be considered, and the cache may use more efficient burst transfers to load the program code into the cache. Furthermore, since cache lookup and replacement only have to be performed at call and return instructions, those cache management operations are not on the critical path of the processor pipeline and could be multi-cycle operations. Finally, the cache analysis only needs to consider a few instructions such as call and return, as all other instructions are guaranteed hits and do not change the cache state.

However, the application code must be partitioned into code blocks at compile time. If the code consists only of small blocks, the method cache may evict entries due to its limited associativity. Large code blocks containing control flow or call sites might be evicted before all instructions in that block have been executed. The compiler must therefore partition the control flow graph into code blocks so that the cache performance is not degraded either by the cache's associativity or by loading unused code into the cache.

A method cache can be implemented in a multitude of ways: *Fixed block method cache*: The cache is organized into blocks of fixed size. Each function is allocated in exactly one cache block. While this organization has the highest fragmentation, it allows for a LRU replacement policy. *Variable block method cache*: Like the fixed-block method cache, the cache is organized in blocks of fixed size. However, a function can be allocated to multiple blocks. This leads to a lower fragmentation, but a LRU policy is complex to implement in hardware. Thus, a FIFO replacement policy is usually employed. *Variable sized method cache*: This variant abandons the internal organization in blocks, thus eliminating internal fragmentation.

```
void m() {
  access(m) /* miss if m is not cached */
  access(a);
  access(m) /* miss if a evicted m */
  access(b);
  access(m) /* miss if b evicted m */
  access(c);
  access(m) /* miss if c evicted m */
}
```

Figure 1 Limits of CHMC: FIFO cache analysis (4-way).

3 Scope-Based Cache Analysis

The two cache analysis techniques that have been extensively studied and implemented in industrial WCET analysis tools are cache hit-miss classification (CHMC) and persistence analysis. The former uses abstract interpretation to compute the set of possible cache states at each instruction in a virtually unrolled and inlined program model [18]. Subsequently, each access to memory is either classified as cache hit, as cache miss, or unknown. This information is finally taken into account when analyzing the timing behavior of the pipeline.

As discussed in [1], if the access to a memory block depends on an unpredictable condition, the CHMC technique fails to classify corresponding cache accesses. CHMC analysis is also less effective for FIFO caches, even though a precise cache state abstraction has been developed [3, 4]. In the example in Figure 1, assuming a cache with associativity four, it is not possible to classify any of the individual accesses to `m` as cache hit. This is because any of the accesses to `m` might be a cache miss, depending on the initial state, although there will be at most one cache miss for `m` in total.

Persistence analysis [1, 9] was conceived to improve the precision of the analysis of LRU caches, and overcome the limitation of CHMC concerning conditional cache accesses. The idea is to classify whether an instruction might suffer a cache miss the first time it is executed, but will hit the cache on subsequent accesses. If this is the case, the cache access is said to be (globally) persistent. Because a cache access might be persistent for the execution of one function, but not the whole program or task, a more useful notion is that of local persistence. An access is said to be persistent with respect to a scope (e.g., the execution of a function), if every time the scope is executed, the access will be a cache miss the first time only. The fact that a persistent cache block will be loaded only on the first access during each execution of the scope is subsequently taken into account during WCET calculation.

Persistence analysis as defined in the literature improves the analysis of LRU caches, but is not applicable to caches using the FIFO replacement policy. Consider again the example in Figure 1: although at most one of the accesses to m is a cache miss, it need not be the first one. Therefore it is necessary to generalize the concept of persistence analysis (*first-miss*) to an analysis that accounts for misses that may occur *at most once*, although not necessarily at the first access. Moreover, in order to obtain a precise analysis, we found that it is necessary to review and refine the concept of persistence scopes. Support for scopes that include some but not all accesses to a memory block (e.g. because of shared library routines) and for scopes that do not correspond to functions or loops (e.g. single-entry regions) are crucial for efficient scope-based analysis of FIFO caches, but have not been considered in previous work.

3.1 Scope-based Cache Analysis Framework

The following discussion considers the general form of the scope-based cache analysis that is applicable for standard set-associative instruction caches and also for different variants of the method cache. We will use the term *memory block* to either denote a block of memory associated with a cache line, or a code block in context of the method cache.

The goal of the scope-based cache analysis is to compute a set of scopes $\mathcal{S}(B)$ for each memory block B, such that (1) every access to the memory block is in at least one scope $S \in \mathcal{S}(B)$ (2) during the execution of any scope $S \in \mathcal{S}(B)$, at most one access to B will be a cache miss. If those two conditions are met, the sum of the execution frequencies of all scopes in $\mathcal{S}(B)$ is an upper bound for the number of cache misses for the memory block B.

In summary, these are the constituents of our analysis:

Scope Graph: The scope graph used by our analysis is an acyclic, hierarchical control-flow representation. Each node in the scope graph represents either a function, a loop or a callsite. The scope graph data structure is essential to ensure the performance of our cache analysis.

Conflict Detection: The analysis builds on a decision procedure that determines whether a memory block needs to be loaded at most once during the execution of a scope. In this case, we say the block is conflict-free with respect to the scope.

Computation of Conflict-Free Scopes: For each memory block, the analysis computes a set of conflict-free scopes. We not only consider functions and loops, but dynamically compute single-entry regions that are conflict-free scopes. The analysis traverses the acyclic scope graph bottom-up, and reuses results from nested scopes to ensure good performance.

IPET model: In the IPET model, we introduce *cache miss* variables, that represent the frequency of cache misses at a certain instruction. Their sum corresponds to the total number of cache misses for one memory block. The total number of cache misses of a memory block is in turn bounded by the frequency of the conflict-free scopes associated with the block.

3.2 Scope Graph

A scope graph is a hierarchical representation of the program's control-flow. Each node represents a set of instruction sequences that correspond to one execution of the program fragment represented by the scope. The root node represents all executions of the program to be analyzed, and each of the instruction sequences represented by a child node is included in one of instruction sequences represented by the parent node. The kind of scope graph used in our analysis is *acyclic*; this is essential to permit a bottom-up analysis. The nodes in a scope graph either correspond to functions, loops or callsites. Scopes that correspond to single-entry regions are not represented in the scope graph, but formed dynamically during analysis (see Section 3.3). Nodes in the scope graph are in turn associated with access graphs that model cache accesses within the scope.

Access graphs are acyclic flow graphs and consist of five different kinds of nodes: The *entry node* and *exit node* of an access graph represent the start and end of any cache access sequence associated with the scope, and *control-flow nodes* represent the beginning of a basic block. Cache accesses that are local to the scope are represented by *memory access nodes*. *Subscope nodes* correspond to the execution of a subscope that is a child in the scope graph. Finally, *back-edge nodes* model transfer to the scopes's entry node, for example to model the back edge of a loop.

Algorithm 1 Scope Set Computation

```
 1: procedure CacheScopeAnalysis(ScopeGraph G)
 2:     for all memory blocks B do
 3:         S[B] ← {}
 4:     end for
 5:     for all nodes N ∈ bottom-up–traversal(G) do
 6:         for all memory blocks B accessed in N do
 7:             if IsConflictingScope(B, N) or
                  N is root node then
 8:                 S[B] ← S[B] ∪ CollectScopes(B, N)
 9:             end if
10:         end for
11:     end for
12:     return S
13: end procedure
```

The construction of the scope graph starts at the scope node that represents the program or task to be analyzed, and processes every function exactly once. First, we compute the strongly connected components (SCC) of the function's CFG. Each trivial SCC corresponds to a basic block, and is replaced by a control-flow node and a sequence of zero or more access graph nodes representing memory accesses and function calls. Non-trivial SCCs are replaced by a subscope node representing the corresponding loop, which is processed in turn.

The access graph of a loop is constructed in a similar way. First, we replace back edges by edges to back-edge nodes that represent transfer of control to the scope entry. This way, every access graph is acyclic, which simplifies the implementation of conflict-detection algorithms. Next, the SCCs of the loop's CFG are computed, and the same procedure as for functions is applied to construct the access graph. In this case, non-trivial SCCs correspond to nested loops. The access graph for a callsite is a simple flow graph that executes one of the subscopes corresponding to functions possibly called at the callsite.

3.3 Computation of Conflict-Free Scopes

Algorithm 1 determines a set of scopes for each memory block B, such that the total number of cache misses of B is bounded by the sum of the frequencies of these scopes. It traverses the scope graph bottom-up, starting at the leaves, and determines for every memory block, whether it is conflict-free with respect to the current scope. If the memory block is not conflict-free, the routine CollectScopes computes conflict-free subscopes.

The goal of CollectScopes is to determine conflict-free single-entry regions in the CFG of the conflicting scope (Algorithm 2). The algorithm traverses the access graph of the scope in topological order, visiting every node exactly once. The procedure assumes the existence of functions NewRegion(v) to create a new single-entry region with header v and AddToRegion(R,v) to add v to region R. Back-edge nodes always form a singleton region, as they correspond to edges back to the entry of the access graph (Line 5). In order to expand a region, all predecessors of a node have to be in the same region (Line 6). Moreover, a node can only be in the same region as its predecessors, if the resulting scope is conflict-free with respect to the memory block. Accesses to memory blocks in conflicting subscopes need not be collected, as they have already been handled while processing the subscope (Line 19).

Note that if the call to AddToRegion(R,v) on Line 11 is replaced by NewRegion(v), one obtains a simpler variant of the algorithm that only considers static persistence scopes.

Algorithm 2 Scope Collection

```
 1: procedure CollectScopes(Block B, Scope N)
 2:     G ← access graph of N
 3:     for all nodes v ∈ topological-traversal(G) do
 4:         if ∃u ∈ preds(v) s.t. u is back-edge node then
 5:             NewRegion(v)
 6:         else if ∃u ∈ preds(v) s.t. ∀w ∈ preds(v)
              GetRegion(u) = GetRegion(w) then
 7:             R ← GetRegion(u)
 8:             if IsConflictingRegion(B, R ∪ {v}) then
 9:                 NewRegion(v)
10:             else
11:                 AddToRegion(R, v)
12:             end if
13:         else
14:             NewRegion(v)
15:         end if
16:     end for
17:     S ← { }
18:     for all regions R do
19:         if B is accessed in R and
              R is not a conflicting subscope then
20:             S ← S ∪ R
21:         end if
22:     end for
23:     return S
24: end procedure
```

3.4 Conflict Detection

So far, the cache analysis framework did not distinguish between different replacement strategies, but relied on the existence of the decision procedures IsConflictingScope and IsConflictingRegion that we describe next.

A memory block is *conflict-free* with respect to a scope, if it will be loaded from memory at most once during the execution of this scope. For set-associative caches using either the LRU or FIFO replacement strategy, this is the case if the cardinality of the set of all distinct cache lines that (1) map to the same cache set and (2) are possibly accessed during the execution of the scope, is less than or equal to the associativity of the cache. For all variations of the method cache, a memory block is conflict-free if (1) the number of distinct code blocks is less than or equal to the associativity of the cache and (2) the total size of all the distinct accessed code blocks, is less than or equal to the size of the method cache.

IsConflictingScope(B, N) and IsConflictingRegion(B, R) are true if the memory block B is *not* conflict-free with respect to the scope associated with N and R, respectively.

The conflict detection strategies described above are sound for both LRU and FIFO caches. The LRU replacement strategy also permits a potentially more precise conflict detection routine, however. In an LRU cache, a memory block is conflict-free with respect to the scope if it is locally persistent with respect to the scope. This in turn is the case if any sequence of accesses between two accesses to the memory block is conflict free.

3.5 IPET Modeling

The integration of the cache miss constraints into the IPET model is shown in Algorithm 3. In the algorithm, we write $f(x)$ to denote a linear expression that corresponds to the execution frequency of x. For all memory access nodes, we introduce a *cache miss variable* that models the frequency of cache misses at this point in the program. The cost of this variable is the cost

Algorithm 3 Extend IPET

```
1: procedure EXTENDIPET(ScopeGraph G)
2:     Scopes ← CACHESCOPEANALYSIS(G)
3:     for all access nodes N do
4:         add cache miss instruction variable N_m
5:         assert N_m ≤ f(N)
6:         cost(N_m) = miss penalty for N
7:     end for
8:     for all memory blocks B do
9:         let A = access nodes for B
10:        assert ∑_{N∈A} f(N_m) ≤ ∑_{S∈Scopes[B]} f(S)
11:    end for
12: end procedure
```

of a cache miss of the corresponding memory block, and is reflected in the objective function of the ILP. Furthermore, each of these variables is obviously bounded by the frequency of the corresponding memory access (Line 5). For all memory blocks potentially accessed, we assert that the sum of all corresponding cache miss variables is bounded by the sum of the frequencies of scopes in the conflict-free scope set for that block (Line 10).

4 Evaluation

The target platform for our evaluation is the Patmos architecture [17]. Patmos uses a five-stage in-order dual-issue RISC pipeline; floating-point operations are performed in software. The `patmos-clang` compiler builds on the LLVM compiler framework, and provides support for WCET analysis, as well as a backend for the Patmos architecture [14]. The benchmarks are compiled using the default optimization settings (-O2), which enables most LLVM optimizations. Measurements were carried out using `pasim`, a cycle-accurate simulation of the Patmos architecture. We do not use a data cache and assume data resides in a scratch-pad memory. In our setting, the relative influence of the instruction cache performance on the performance of the processor is thus more pronounced. Since Patmos is a time-composable architecture, this decision does not have any effect on the timing of the instruction cache.

For the memory timing, we use the actual timings of an SDRAM controller developed for Patmos [10]. This memory controller is able to hide latencies for longer burst lengths. In the evaluation we use burst sizes of 8 words that take 11 cycles for a read.

The WCET analysis is carried out using `platin`, a tool that is bundled with our compiler and is primarily intended to bridge the gap between compilers and WCET analysis tools. For evaluation purposes, `platin` supports the extraction of flow fact hypothesis from test runs. This is useful for early-stage development estimates, and provides an excellent way to compare cache analysis results with measurements [7]. The cache analysis itself is only provided with information on infeasible paths, whereas the WCET calculation uses all flow facts that were extracted from test the runs, including the observed frequency of basic blocks.

For our evaluation, we considered benchmarks from two well-established benchmark suites for WCET analysis; the MRTC benchmark suite [6] and PapaBench [13]. In this section, we compare the analysis results of a set-associative instruction cache and three variations of the method cache: (1) a set-associative cache with LRU replacement, (2) a fixed block method cache, using LRU replacement, (3) a variable block method cache with FIFO replacement, and (4) a variable size method cache with FIFO replacement.

Table 1 shows the evaluation results for 1 KB caches, using our scope-based cache analysis technique. We chose rather small 1 KB caches to stress test the analysis, as using larger

Table 1 Comparison of an instruction cache and three different method cache variants.

	benchmark	I$-8	FB-4	VB-8	VS-8	VS-8-NS
1	ndes	1.00	1.07	1.07	*1.05*	1.24
2	jfdctint	1.00	1.11	1.11	1.11	*1.00*
3	cnt	1.00	1.08	1.08	1.08	*1.00*
4	fdct	1.00	1.09	1.09	1.09	*1.00*
5	adpcm	1.00	1.01	1.01	1.01	1.01
6	edn	1.00	1.06	1.06	*1.06*	1.13
7	select	1.00	1.09	1.09	*1.11*	1.35
8	fbw/send_to_pilot	1.00	1.18	1.13	*1.09*	1.59
9	qsort	1.00	1.75	1.67	*1.12*	5.55
10	fbw/check_failsafe	1.00	1.60	1.60	*1.31*	1.81
11	fbw/check_values	1.00	1.60	1.60	*1.31*	1.82
12	ud	1.00	1.08	1.07	*1.02*	1.12
13	fbw/ppm_task	1.00	1.44	*1.39*	1.40	1.85
14	nsichneu	1.00	1.18	1.18	*1.18*	1.29

caches results in many of the benchmarks being trivial to analyze. For all method cache configurations but the last one, we used a reasonable default for the function splitter, which prefers block of roughly 256 bytes if possible. The baseline is the performance of a standard 8-way set-associative cache (`I$-8`). The next column provides the relative WCET performance of a fixed-block method cache (`FB-4`). We chose a 4-way cache here to have reasonable large block sizes (256 bytes) that functions need to be split to. The fourth column (`VB-8`) shows the performance of variable-block method cache with a block size of 128 bytes.

The column `VB-8` shows the performance of variable-block method cache with associativity 8. The third method-cache variant is a variable-sized cache, where code blocks do not need to be multiples of some block size. The result for this cache using the usual function splitter setting is shown in column `VS-8`. Additionally, we evaluated the performance of the variable size method cache when the function splitter only splits functions down to fit in the cache (`VS-8-NS`).

It can be seen that the method cache performs equal or poorer than the instruction cache in all explored configurations. However, for most benchmarks the increase of the WCET bound is in an acceptable range for the variable-sized block method cache configuration. What is surprising is that the very simplistic fixed-block cache performs quite well due to the good compiler support through function inlining and splitting. This might be an indication that there is more headroom to adapt the function splitter to provide better results for the variable-sized method cache.

5 Related Work

The analysis presented in this paper builds on our earlier work on method cache analysis for JOP [8]. The previous analysis was tied to the concept of a Java method, did not use the scope-graph representation, and only considered methods as scopes. Metzlaff and Ungerer also compared the WCET bounds for different instruction cache architectures [12]. For the method cache, their CHMC analysis represents possible cache states as the powerset of all possible cache configurations. As this is prohibitively expensive if the number of possible cache configurations grows, the cache has to be reset at the analysis start, and might not be applicable to larger applications [11].

For set-associative caches, the CHMC analysis of Grund and Reineke [4] exploits the fact that when memory blocks are accessed repeatedly, it becomes possible to classify accesses in a FIFO cache. Their work was the first to demonstrate that CHMC analysis is feasible for FIFO caches. An innovative approach by the same authors is to use results from cache competitive analysis [15]. The idea of this analysis is to determine the maximum number of misses for a smaller LRU cache that is known to perform no better than the FIFO cache. However, this analysis effectively reduces the size of the cache visible to the analysis by a factor of two.

Whereas early persistence analyses were unsound, recent articles published corrected dataflow algorithms that promise improvements compared to CHMC analyses [9, 1]. The implementation in [1] combines persistence analysis and CHMC, which appears to be an advisable strategy in practice, especially when dealing with complex architectures. Neither article, however, considers the FIFO replacement strategy, fine-grained scopes or memory blocks that belong to more than one scope.

Guan et. al. recently presented an approach to FIFO cache analysis that, similar to ours, derives cache miss constraints [5]. Instead of precise scopes, they attempt to improve precision by deriving *cache miss ratios* relative to the scope entry. Their evaluation demonstrates significant improvements compared to a CHMC-based FIFO analysis, and could possibly be integrated into our framework.

6 Conclusion

In this paper, we presented a new cache analysis framework that generalizes work on cache persistence analysis and it is applicable to a wide range of instruction caches using either FIFO or LRU replacement strategies.

We used this cache analysis and our LLVM-based compiler to explore method caches in a realistic setting. We found that even in combination with a memory controller that is limited to bursts of fixed length, our analysis is able to attest the method cache a WCET performance that is competitive to an 8-way set-associative instruction cache. Furthermore, we showed that adding explicit support for the method cache in the compiler has a high impact on the performance of the method cache.

In future work, we plan to enhance the function splitter algorithm and integrate it with the cache analysis so that the cache block region formation can profit from the knowledge of the analysis. Finally, we also plan to apply the scope-based cache analysis to data caches, where persistence analysis has been proven to be useful before.

References

1. Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computer Systems*, 12(1s):40, 2013.
2. Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *to appear: Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, Reno, Nevada, USA, June 2014. IEEE.
3. Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS'09, pages 120–136, Berlin, Heidelberg, 2009. Springer-Verlag.

4 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS'10, pages 155–164, 2010.

5 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. Fifo cache analysis for wcet estimation: A quantitative approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'13, pages 296–301, San Jose, CA, USA, 2013. EDA Consortium.

6 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

7 Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. Towards an open timing analysis platform. In *11th International Workshop on Worst-Case Execution Time Analysis*, July 2011.

8 Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 23–34, Dublin, Ireland, July 2009. OCG.

9 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

10 Edgar Lakis and Martin Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

11 Stefan Metzlaff and Theo Ungerer. Impact of instruction cache and different instruction scratchpads on the wcet estimate. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1442 –1449, june 2012.

12 Stefan Metzlaff and Theo Ungerer. A comparison of instruction memories from the WCET perspective. *Journal of Systems Architecture*, 60(5):452–466, 2013.

13 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. PapaBench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

14 Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

15 Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES'08, pages 51–60, 2008.

16 Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

17 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

18 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis

Sahar Abbaspour[1], Alexander Jordan[1], and Florian Brandner[2]

1 **Department of Applied Mathematics and Computer Science**
Technical University of Denmark `{sabb,alejo}@dtu.dk`
2 **Computer Science and System Engineering Department**
ENSTA ParisTech `florian.brandner@ensta-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. A stack cache, for instance, allows the compiler to efficiently cache a program's stack, while static analysis of its behavior remains easy. Likewise, its implementation requires little hardware overhead.

This work introduces an optimization of the standard stack cache to avoid redundant spilling of the cache content to main memory, if the content was not modified in the meantime. At first sight, this appears to be an average-case optimization. Indeed, measurements show that the number of cache blocks spilled is reduced to about 17% and 30% in the mean, depending on the stack cache size. Furthermore, we show that lazy spilling can be analyzed with little extra effort, which benefits the worst-case spilling behavior that is relevant for a real-time system.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

Keywords and phrases Lazy Spilling, Stack Cache, Real-Time Systems, Program Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2014.83

1 Introduction

In order to meet the timing constraints in systems with hard deadlines, the worst-case execution time (WCET) of real-time software needs to be bounded. This WCET bound should never underestimate the execution time and should be as tight as possible. Many features of modern processor architectures, such as pipelining, caches, and branch predictors, improve the average performance, but have an adverse effect on WCET analysis. Time-predictable computer architectures propose alternative designs that are easier to analyze.

Memory accesses are crucial for performance. This also applies to time-predictable alternatives. Analyzable memory and cache designs as well as their analysis thus recently gained considerable attention [13, 8, 9]. One such alternative cache design is the *stack cache* [1, 5], i.e., a specialized cache dedicated to stack data, which is intended as a complement to a regular data cache. This design has several advantages. Firstly, the number of accesses going through the regular data cache is greatly reduced, promising improved analysis precision. For instance, imprecise information on access addresses can no longer interfere with the analysis of stack accesses (and vice verse). Secondly, the stack cache design is simple and thus easy to analyze [5]. The WCET analysis of traditional caches requires precise knowledge about the addresses of accesses [13] and has to take the (from an analysis point of view

© Sahar Abbaspour, Alexander Jordan, and Florian Brandner;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 83–92

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

complex) replacement policy into account. The analysis of the stack cache on the other hand is much easier and amounts to a simple analysis of the cache's fill level (*occupancy*) [5].

In this paper we propose an optimization to improve the performance of the stack cache's reserve operation based on the following observation: in many cases, the actual data spilled by a reserve has exactly the same value as the data already stored in main memory. This may happen in situations where data is repeatedly spilled, e.g., due to calls in a loop, but not modified in the meantime. Thus, the main idea is to track the amount of data that is coherent between the main memory and the cache. The cache can then avoid to needlessly spill coherent data. We show that this tracking can be realized efficiently using a single pointer, the so-called *lazy pointer*. We furthermore show that the existing analysis [5] can be easily adapted to this optimization, by simply refining the notion of occupancy, to account for the coherent space defined by the lazy pointer.

Section 2 introduces the stack cache, followed by a discussion of related work. Section 4 presents the motivation for our work. In Section 5, we explain lazily spilling and its static analysis. We finally present the results from our experiments in Section 6.

2 Background

The original stack cache [1] is implemented as a kind of ring buffer with two pointers: *stack top* (`ST`) and *memory top* (`MT`). The former points to the top of the logical stack, which consists of all the stack data that is either stored in the cache or main memory. The latter points to the top element present in main memory only. For simplicity, we herein assume a hardware implementation[1] with a stack growing towards lower addresses.

The difference $\texttt{MT} - \texttt{ST}$ represents the amount of occupied space in the stack cache. Clearly, this value cannot exceed the total size of the stack cache's memory $|SC|$, thus $0 \leq \texttt{MT} - \texttt{ST} \leq |SC|$. The stack control instructions hence manipulate the two stack pointers, while preserving this equation, and initiate the corresponding memory transfers as needed. A brief summary is given below (details are available in [1]):

sres x**:** Subtract x from `ST`. If this violates the equation from above, i.e., the stack cache size is exceeded, a *spill* is initiated, which lowers `MT` until the invariant is satisfied again.

sfree x**:** Add x to `ST`. If this results in a violation of the invariant, `MT` is incremented accordingly. Memory is not accessed.

sens x**:** Ensure that the occupancy is larger than x. If this is not the case, a *fill* is initiated, which increments `MT` accordingly so that $\texttt{MT} - \texttt{ST} \geq x$ holds.

Using these three instructions, the compiler generates code to manage the stack frames of functions, quite similar to other architectures with exception of the ensure instruction. For brevity, we assume a simplified placement of these instructions. Stack frames are allocated upon entering a function (`sres`) and freed immediately before returning from a function (`sfree`). The content of a function's stack frame might be evicted from the cache upon function calls. The compiler thus ensures a valid stack cache state, immediately after each call site (`sens`). This simplified placement can be relaxed as discussed in Section 5.2. We furthermore restrict functions to only operate on their locally allocated stack frames. Any stack data shared between functions, or exceeding the stack cache's size, is allocated on the so-called *shadow stack* outside the stack cache.

---

[1] Note that the stack control instructions could also be implemented by means of software.

3 Related Work

Static analysis [12, 3] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. Through its simpler analysis model, the stack cache does not require the precise knowledge of addresses, thus eliminating a source of complexity and imprecision. It has been previously shown that the stack cache serves up to 75% of the dynamic memory accesses [1]. Our approach to compute the worst-case behavior of the stack cache has some similarity to techniques used to statically analyze the maximum stack depth [2]. Also related to the concept of the stack cache, is the register-window mechanism of the SPARC architecture, for which limited WCET analysis support exists in Tidorum Ltd.'s Bound-T tool [11, Section 2.2].

Alternative caching mechanisms for program data exist with the Stack Value File [6] and several solutions based on Scratchpad Memory (SPM) (e.g. [7]), which manage the stack in either hardware or software.

4 Motivating Example

Figure 1 shows a function `bar` using the stack cache allocating a stack frame of two words (l. 2) on a stack cache of size of 8 words. The stack frame is freed before returning from the function (l. 16). The loop (l. 5–14), repeatedly calls function `foo`. We assume function `foo` reserves 8 words, thus evicts the entire stack cache content and displaces 8 words.

Assuming no stack data has been allocated so far, the stack cache is entirely empty. The `MT` and `ST` pointers are pointing to the top of the stack at address 256. The `sres` instruction of function `bar` (l. 2) decrements `ST` by two, without any spilling. At this point, `MT` points to address 256 and `ST` to 248 resulting in an occupancy of 2 words.

The stack store and load instructions (l. 4, 7) do not modify the `MT` and `ST` pointers. After the function call to `foo`, an `sres` execution reserves 8 words. Decrementing `ST` by 32 with an unmodified `MT` would result in an occupancy of 10 words $(256 - 216)$, exceeding the stack cache size. Two words are thus spilled and `bar's` stack frame is transferred to the main memory (addresses [248, 255]). Before returning from `foo`, its stack frame is freed, by incrementing `ST` by 32 (`ST` = `MT` = 248), meaning that the stack cache is empty. The `sens` instruction (l. 11) executed next, is required to ensure that `bar's` stack frame is present in the cache for the load of the next loop iteration. Therefore, a cache fill operation increments the `MT` pointer to 256. The current cache state is identical to the state before the function call. For subsequent loop iterations, the stack cache states change in exactly the same way. An interesting observation at this point is that the values of the respective stack slots are loop-invariant and do not change. Consequently, during every iteration the exact same values

```
1  function bar()
2     sres 2
3     // store loop-invariant stack data
4     sws [1] = ...
5  loop:
6     // load loop-invariant stack data
7     lws ... = [1]
8     // displaces entire stack cache
9     call foo
10    // reload local stack frame
11    sens 2
12    cmp ...
13    // jump to beginning of loop
14    bt loop
15    // exit function
16    sfree 2
17    ret
```

Figure 1 `foo` evicts the entire stack cache, `bar`'s stack data is thus spilled on each iteration.

are written to the main memory. Even more, the respective memory cells already hold these values for all loop iterations except for the first. A standard stack cache obviously lacks the means to avoid this redundant spilling of stack data that is known to be *coherent*.

5 Lazy Spilling

We propose to introduce another pointer, which we call *lazy pointer* (`LP`), that keeps track of the stack elements in the cache that are known to be coherent with main memory. The lazy pointer only refers to the data in the stack cache, thus the invariant $\texttt{ST} \leq \texttt{LP} \leq \texttt{MT}$ holds.

The `LP` divides the reserved space in the stack cache into two regions: (a) a region between the `ST` and `LP` and (b) a region between the `LP` and `MT`. The former region defines the *effective occupancy* of the stack cache, i.e., it contains potentially modified data. The data of the second region is coherent between the main memory and the stack cache. Whenever the `ST` moves up past the `LP` or the `MT` moves down below the `LP`, the `LP` needs to be adjusted accordingly. When a stack store instruction writes to an address above the `LP`, some data potentially becomes incoherent. Hence, the `LP` should move up along with the effective address of the store.

5.1 Implementation

Following the above observations, we need to adapt the stack control instructions and the stack store instructions to support lazy spilling. Note, however, that lazy spilling does not impact the `ST` and the `MT`, i.e., their respective values are identical to a standard stack cache.

`sres` x: The stack cache's `sres` instruction decrements the `ST`. Moreover, the reserve potentially spills some stack cache data to the main memory and thus may decrement the `MT`. To respect the invariant from above, this may require an adjustment of the `LP` to stay below the `MT`. We can, in addition, exploit the fact that the newly reserved cache content is uninitialized – and thus can be treated as coherent with respect to the main memory. For instance, when $\texttt{ST} = \texttt{LP}$ before the reserve, all the stack cache content is known to be coherent, which allows us to retain the equality $\texttt{ST} = \texttt{LP}$ even after the `sres` instruction. Moreover, when the space allocated by the current `sres` covers the entire stack cache, all the data in the stack cache is uninitialized. In this case it is again safe to assume $\texttt{ST} = \texttt{LP}$ after the reserve. Apart from updating the `LP`, the spilling mechanism itself requires modifications. Originally, the `MT` pointer is used to compute the amount of data to spill. However, when lazy spilling is used, the amount of data to spill does not depend on the `MT` anymore. Instead, the `LP` has to be used to account for the coherent data present in the stack cache. With respect to spilling, the `LP` effectively replaces the `MT` – hence the term effective occupancy for the region between the `ST` and the `LP`.

`sfree` x: The stack cache's `sfree` instruction increments the `ST`. The `MT` may potentially increase as well. To satisfy the invariant from above, the `LP` is incremented in case it is below the `ST`. No further action is required.

`sens` x: The stack cache's `sens` instruction does not modify the `ST` and may only increase the `MT`. The invariant is thus trivially respected. Moreover, coherency of the data loaded into the cache is guaranteed. The `sens` instruction thus requires no modification.

`store`: The stack store instruction writes to the stack cache and may thus change the value of previously coherent data. Therefore, the `LP` needs an adjustment whenever the effective address of the store is larger than the `LP`, i.e., the `LP` needs to be greater than the effective address of the store instruction.

```
Result: Construct SCA Graph G
initialize worklist W with entry context (s, 0);
while unhandled context c in W do
    f ← function of c;
    e ← effective occupancy of c;
    foreach call instruction i in f calling f′ do
        e′ ← min(e + r, o_eff[i]);
        c′ ← (f′, e′);
        add edge c → c′ to G;
        if c′ is a new context then
            add new node to G;
            add context c′ to W
```

Algorithm 1: Construct Spill Cost Analysis Graph

▶ **Example 1.** Consider again the program from Figure 1. The stack cache is initially empty and the ST, the MT, and the LP point to address 256. As before, the sres instruction moves the ST down to address 248. The LP moves along with the ST, since ST = LP and we assume that the newly reserved and uninitialized space is coherent with the main memory. Next, the store instruction (l. 4) modifies some data present in the stack cache. The LP consequently has to move upwards and now points to address 252. The first call to function foo causes two words to be spilled (see Section 4) and the MT is thus decremented to 248. As the LP is larger than the MT at this point, the LP would normally require an adjustment. However, since we assume that foo reserves the entire stack cache, it is safe to set the LP to the value of ST (216). Any stores within foo then automatically adjust the LP as needed. The stack cache is again empty after returning from function foo. All three pointers of the stack cache then point to address 248. The sens reloads the stack frame of function bar (l. 11), leaving both ST and LP unmodified, but incrementing MT to 256. The occupancy of the normal stack cache at this point is 8 (2 words), while the effective occupancy of the stack cache with lazy spilling is 0, i.e., the entire cache content is known to be coherent. The effective occupancy for the next call to foo as well as for subsequent iterations of the loop, stays at 0. The reserve within foo thus finds the entire stack cache content coherent and avoids spilling completely. Finally, when the program exits the loop, the sfree instruction (l. 16) increments ST to 256. In order to satisfy the invariant, LP also has to be incremented.

5.2 Static Analysis

With the restrictions from Section 2 in place, the analysis algorithms assume that reserve and free instructions appear at a function's entry and exit; this may be relaxed, given the program remains well-formed regarding its stack space. Due to space restrictions, we refer to [5], which describes the relaxation and the analysis of the original stack cache design.

The filling behavior of ensure instructions is not affected by lazy spilling, therefore, the context-insensitive *ensure analysis* remains unchanged compared to its original. The spilling behavior of a reserve instruction depends on the state of the stack cache at function entry, which in turn depends on a nesting of stack cache states of function calls reaching the current function. To fully capture this context-sensitive information, we can represent spill cost in the *spill cost analysis graph* (SCA graph). An example SCA graph is depicted in Figure 2b. The effective spill cost for an sres instruction in a specific calling context is computed from the occupancy of the context and the locally reserved space x (in sres x). This value is then multiplied with the timing overhead of the data transfer to external memory.

Without lazy spilling, the construction of the SCA graph depends on the notion of stack cache *occupancy*, i.e., the utilized stack cache region of size MT − ST. In order to benefit from the reduced overhead of lazy spilling, we need to consider the possibly smaller region LP − ST. Bounds for this effective occupancy of the stack cache are propagated through the call graph during *reserve analysis*, which constructs the SCA graph as shown in Algorithm 1. The o_{eff}-bound used by the algorithm improves the overestimation still present in context-sensitive reserve analysis. It can be computed by an intra-procedural data-flow analysis, which considers *minimum displacement* and the destinations of stack cache stores. For every point within a function, but most interestingly before calls, o_{eff} represents a local upper bound on the effective occupancy.

As a pre-requisite for the intra-procedural analysis, minimum displacements for all functions in the program are computed. This is achieved by several shortest-path searches in the program's call graph. The resulting values are bounded by the size of the stack cache $disp(i) = \min(|SC|, d_{min}(i))$ and used in the data-flow transfer functions of the analysis:

$$OUT(i) = \begin{cases} \min(IN(i), |SC| - disp(i)) & \text{if } i = call \quad \text{(1a)} \\ \max(IN(i), blocks(n)) & \text{if } i = \texttt{sws n} \quad \text{(1b)} \\ IN(i) & \text{otherwise} \quad \text{(1c)} \end{cases}$$

After returning from a call instruction i, the LP cannot have moved upwards with respect to its location before the call. In fact, a function call can only leave the LP unchanged or move it downwards. I.e., if stack contents of the current function are spilled, this potentially increases the coherent region of the stack cache and thus decreases the effective occupancy (1a). The amount of spilling is represented by the previously computed (worst-case) minimum displacement $disp(i)$.

Opposite to calls, the LP moves up when i is a store instruction (1b). Since the number of data words per stack cache block is fixed, the number of blocks that become incoherent due to a store, only depends on its destination n (in $blocks(n) = \lceil n/\text{words-per-block}\rceil$).

Note that while an `sens` instruction can impact the original analysis of occupancy bounds, it does not influence the effective occupancy and thus plays no role during reserve analysis in the presence of lazy spilling.

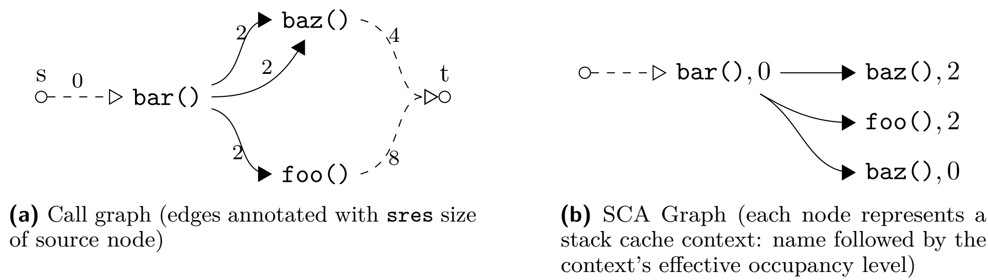
In order to safely initialize the o_{eff} bound and propagate it between instructions (i.e., from the OUT-values of the predecessors of an instruction to its IN-value (2b)), we define the following transfer functions:

$$IN(i) = \begin{cases} |SC| & \text{if } i \text{ is entry} \quad \text{(2a)} \\ \max\limits_{p \in Preds(i)} (OUT(p)) & \text{otherwise} \quad \text{(2b)} \end{cases}$$

It is further worth noting that in practice not all stack store instructions need to be analyzed. As long as the effective occupancy value exceeds the stack cache size reserved by the current function (i.e., the coherent region does not overlap with the current stack frame), a particular store has no effect on the analysis state.

▶ **Example 2.** Consider the program from Figure 1, where two additional function calls to function `baz` are performed. The first call appears right before entering the loop, while the other appears right after exiting the loop.

The call graph for this program is shown in Figure 2a. This graph also shows the size of the stack frame reserved by each function on the outgoing call edges. For functions without calls (`baz` and `foo`), call edges to an artificial sink node t are introduced. Using these annotations, the minimum and maximum displacement of each function can be determined



(a) Call graph (edges annotated with `sres` size of source node)

(b) SCA Graph (each node represents a stack cache context: name followed by the context's effective occupancy level)

Figure 2 Call graph and SCA Graph for the program of Example 2.

using a shortest and longest path search respectively. The minimum displacements for the three functions are 8, 6, and 4; for `foo`, `bar`, and `baz` respectively.

The analysis of this program results in the SCA graph shown by Figure 2b. The analysis starts with an empty stack cache and finds that `bar` allocates 2 words (8 bytes), initializes the respective stack slots and then calls `baz`. Since `bar`'s stack content was not spilled before, the effective occupancy when entering `baz` evaluates to 2. Assuming a stack cache size of 8 words, the function can be executed without spilling. The analysis thus can deduce that the `LP` was not modified with regard to `bar`'s stack frame. Propagating this information to the function call of `foo` within the loop results in a context for `foo` with the same occupancy of 2. Once the program exits the loop, another call to `baz` is executed. Due to `foo` displacing the full stack cache and the lack of relevant stores, the data-flow analysis is able to propagate the effective occupancy of 0 down to this call. This is reflected in the SCA graph by containing a node for `baz` annotated with its effective 0-occupancy.

6 Experiments

We use the Patmos processor [10] to evaluate the impact of lazy spilling on hardware costs, average program performance, and worst-case spilling bounds. The hardware model of the processor was extended and statistics were collected on the speed and resource requirements after synthesis (Altera Quartus II 13.1, for Altera DE2-115). The average performance measurements were performed using the MiBench [4] benchmarks suite. The programs were compiled using the Patmos LLVM compiler (version 3.4) with full optimizations (`-O3`) and executed on a cycle-accurate simulator. We compare five configurations utilizing (a) a standard data cache combined with a lazily spilling stack cache having a size of 128 or 256 bytes (LP_{128}, LP_{256}), (b) a standard data cache combined with a standard stack cache (SC_{128}, SC_{256}), and (c) a standard data cache alone (DC). The stack caches perform spilling and filling using 4 byte blocks. The data cache is configured to have a size of 2 KB, a 4-way set-associative organization, with 32 byte cache lines, LRU replacement, and a write-through strategy (recommended for real-time systems [13]). In addition to data caches, the simulator is configured to use a 16 KB method cache for code. The main memory is accessed in 16 byte bursts and 7 cycles latency.

6.1 Implementation Overhead

The standard stack cache of Patmos is implemented by a controller, which executes spill and fill requests, and control instructions (`sres`, `sfree`, `sens`) sending requests to that controller. The controller is independent from the extensions presented here. It thus suffices to extend the `sres`, `sfree`, and stack store instruction as indicated in Section 5.1. The Patmos processor

has a five-stage pipeline (fetch, decode, execute, memory access, write back). The stack cache reads the `ST` and the `MT` in the decode stage and immediately computes the amount to spill/fill. The `LP` is read in the decode stage as well, which allows us to perform all `LP`-related updates in the decode and execute stages. That way additional logic on the critical path in the memory stage, where the cache is actually accessed, is avoided. This applies in particular to the store instruction, whose effective address only becomes available in the execute stage. For lazy spilling, a single additional register is needed (`LP`). Updating the `LP` in the `sres` instruction adds two multiplexers to the original implementation. The `sfree` and stack store instructions each need an additional multiplexer. The area overhead is, therefore, very low. Moreover, these changes do not affect the processing frequency.

6.2 Average Performance

Table 1 (columns Spill) shows the reduction in the number of blocks spilled in comparison to the standard stack cache. Note that results for `rawcaudio` and `rawdaudio` are not shown, as they never spill due to their shallow call nesting depth. The best result for LP_{128} is achieved for `bitcnts`, where lazy spilling practically avoids spilling. In the mean, spilling is reduced to just 17%. The worst result is attained for `qsort-small`, where 62% of the blocks are spilled. For LP_{256} spilling is reduced to 30% in the mean. The best result is observed for `search-large`, where essentially all spilling is avoided. For `crc-32`, `drijndael`, `erijndael`, and `sha` only marginal improvements are possible, since these benchmarks already spill little in comparison to the other benchmarks. The worst result of those benchmarks with a relevant amount of spilling is obtained for `qsort-small`, where 76% of the blocks are spilled.

Miss rates are not suitable for comparison against standard data caches. We thus compare the number of bytes accessed by the processor through a cache in relation to the number of stall cycles it caused, i.e., $\frac{\#RD+\#WR}{\#Stalls}$. A high value in Table 1 (columns SC and DC) means that the cache is efficient, as data is frequently accessed without stalling. The data cache alone gives values up to 3.3 only. Ignoring benchmarks with little spilling, the best result for SC_{128} is achieved by `dbf` (477.4). For SC_{256}, `bitcnts` gives the best result (17054.7). Lazy

Table 1 Bytes accessed per stall cycle and reduction in spilling for the various configurations.

	SC_{128}			LP_{128}		SC_{256}			LP_{256}		DC
Benchmark	SC	DC	Spill	LP-SC	DC	SC	DC	Spill	LP-SC	DC	
basicmath-tiny	2.3	1.1	0.17	4.0	1.1	26.4	1.1	0.53	34.0	1.1	1.1
bitcnts	4.6	191.6	0.00	12.2	191.6	17054.7	193.7	0.71	19201.4	193.7	1.2
cjpeg-small	116.9	1.0	0.51	148.4	1.0	3470.7	1.0	0.09	6154.4	1.0	1.1
crc-32	9.0	0.9	0.03	21.3	0.9	814.9	0.9	1.00	814.9	0.9	0.9
csusan-small	11.3	2.2	0.16	18.6	2.2	1218.8	2.3	0.72	1430.0	2.3	1.5
dbf	477.4	1.0	0.47	623.0	1.0	–	1.0	–	–	1.0	1.0
dijkstra-small	19.5	1.4	0.20	32.8	1.4	335.2	1.4	0.54	433.7	1.4	1.4
djpeg-small	9.0	0.8	0.34	13.5	0.8	293.4	0.8	0.66	361.5	0.8	0.8
drijndael	15.8	0.9	0.20	28.7	0.9	185620.0	0.9	1.00	185620.0	0.9	0.9
ebf	172.5	1.0	0.44	224.6	1.0	–	1.0	–	–	1.0	1.0
erijndael	32.6	0.9	0.57	43.3	0.9	258340.0	0.9	1.00	258340.0	0.9	0.9
esusan-small	15.9	3.4	0.25	25.3	3.4	70.7	3.6	0.02	139.5	3.6	1.5
fft-tiny	3.1	1.1	0.08	5.8	1.1	85.0	1.1	0.56	103.4	1.1	1.1
ifft-tiny	3.1	1.2	0.08	5.9	1.2	83.1	1.1	0.56	101.0	1.1	1.1
patricia	2.5	1.0	0.27	4.2	1.0	26.4	1.0	0.55	31.9	1.0	1.0
qsort-small	3.1	1.0	0.62	3.7	1.0	7.8	1.0	0.76	8.6	1.0	1.0
rsynth-tiny	16.0	1.9	0.08	29.9	1.9	1096.1	1.9	0.48	1539.8	1.9	1.3
search-large	2.9	0.8	0.48	3.9	0.8	26.3	0.8	0.00	52.5	0.8	0.9
search-small	2.9	0.8	0.49	3.7	0.8	28.1	0.8	0.02	54.8	0.8	0.9
sha	8.3	1.6	0.20	14.1	1.6	668.7	1.6	0.91	700.6	1.6	1.6
ssusan-small	29.2	17.1	0.20	43.9	17.1	4313.5	17.1	0.80	4678.0	17.1	3.3

spilling leads to consistent improvements over all benchmarks. An interesting observation is that for most benchmarks the presence of a stack cache *improves* the performance of the data cache. The best example for this is `bitcnts`, but also `csusan` and `ssusan` profit considerably, where the data cache alone delivers 1.2 bytes per stall cycle. When a stack cache is added to the system, this value jumps up to 192.4 and 196.1 respectively.

Our measurements show that lazy spilling eliminates most spilling in comparison to a standard stack cache. Also the efficiency of the standard data cache is improved in many cases. Due to the low memory latency assumed here, this translates to run-time gains of up to 21.8% in comparison to a system with a standard stack cache. In the mean, the speedup amounts to 8% and 9.2% in comparison to a system only equipped with a data cache.

6.3 Static Analysis

We evaluate the impact of lazy spilling on the static analysis by comparing its bounds on the worst-case spilling with the actual spilling behavior observed during program execution. In order to be considered *safe*, the analysis' bounds always need to be larger or equal to the observed spilling of an execution run. Apart from a safe result, the analysis should also provide *tight* bounds. Note that the observed spilling relates to the average-case behavior of the programs, which does not necessarily trigger the worst-case stack cache behavior (the same inputs as in Section 6.2 have been used). However, we still report the estimation gap between worst case and average case, to show the effect of lazy spilling in the analysis.

We extended the Patmos simulator and first verified that the actual spilling of all `sres` instructions executed by a benchmark program is safely bounded by the analysis. We then measured the maximum difference between statically predicted and observed spilling (Max-Spilling-Δ) for each `sres` in every of its contexts (i.e., considering all nodes of the SCA graph reachable by execution). Table 2 shows the results for the 128-byte stack cache configuration and allows for a comparison between statically predicted and dynamically observed spill costs. A first look reveals that static spill cost is reduced for all programs in our benchmark set (down to 12% for `rsynth-tiny`). Furthermore, when the estimation gap is initially low, lazy spilling tends to widen the gap between static and dynamic spill cost

Table 2 Analysis precision: static worst-case compared to observations from dynamic execution (bytes spilled based on maximum spilling per stack cache context)

	SC_{128} Max-Spilling-Δ			LP-SC_{128} Max-Spilling-Δ		
Benchmark	Static	Dynamic	Gap	Static	Dynamic	Gap
basicmath-tiny	68,128	32,040	2.13×	10,052	8,080	1.24×
bitcnts	892	684	1.30×	768	320	2.40×
crc-32	844	652	1.29×	684	372	1.84×
csusan	5,404	2,592	2.08×	2,420	1,196	2.02×
dbf	684	456	1.50×	564	324	1.74×
dijkstra-small	10,220	5,796	1.76×	6,676	2,608	2.56×
drijndael	1,172	664	1.77×	1,024	488	2.10×
ebf	684	456	1.50×	564	324	1.74×
erijndael	880	400	2.20×	752	292	2.58×
esusan	4,724	1,888	2.50×	2,256	1,024	2.20×
fft-tiny	32,484	9,476	3.43×	5,804	3,712	1.56×
ifft-tiny	32,224	9,256	3.48×	5,620	3,548	1.58×
patricia	1,996	1,672	1.19×	1,804	984	1.83×
qsort-small	3,804	1,492	2.55×	2,432	840	2.90×
rsynth-tiny	109,864	15,320	7.17×	13,504	3,140	4.30×
search-large	840	740	1.14×	668	340	1.96×
search-small	828	728	1.14×	708	312	2.27×
sha	1,160	660	1.76×	1,032	448	2.30×
ssusan	6,608	1,824	3.62×	2,452	1,060	2.31×

slightly. But for those benchmarks that exhibit large estimation gaps with a standard stack cache, lazy spilling can even improve analysis precision.

7 Conclusion

From our experiments, we conclude that the benefits of lazy spilling extend from the average case performance to worst-case behavior, where it can even benefit analysis precision. In future work, we plan to introduce loop contexts and a limited notion of path-sensitivity to the analysis, to better capture occupancy states and thus spilling behavior.

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

References

1 S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Techn. for Embedded and Ubiquitous Systems*. IEEE, 2013.
2 Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static memory and timing analysis of embedded systems code. In *Proc. of Symposium on Verification and Validation of Software Systems*, pages 153–163. Eindhoven Univ. of Techn., 2007.
3 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.
4 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 *Proc. of the Workshop on Workload Characterization*, WWC'01, 2001.
5 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*, pages 55–64. ACM, 2013.
6 Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Gary S. Tyson, and Chris J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. of the International Symposium on High-Performance Computer Architecture*, HPCA'01, pages 5–14. IEEE, 2001.
7 Soyoung Park, Hae woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *In Proc. of the Design, Automation Test in Europe Conference*, DATE'07, pages 1–6. ACM, 2007.
8 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 Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108. ACM, 2011.
9 Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.
10 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, volume 18, pages 11–21. OASICS, 2011.
11 BoundT Time and Stack Analyzer – Application Note SPARC/ERC32 V7, V8, V8E. Technical Report TR-AN-SPARC-001, Version 7, Tidorum Ltd., 2010.
12 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS'97, pages 192–203, 1997.
13 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. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.

Identifying Relevant Parameters to Improve WCET Analysis*

Jakob Zwirchmayr[1], Pascal Sotin[1], Armelle Bonenfant[1], Denis Claraz[2], and Philippe Cuenot[2]

1 **Université de Toulouse, IRIT, France**
2 **Continental Automotive France SAS, Toulouse, France**

Abstract

Highly-configurable systems usually depend on a large number of parameters imposed by both hardware and software configuration. Due to the pessimistic assumptions of WCET analysis, if left unspecified, they deteriorate the quality of WCET analysis. In such a case, supplying the WCET analyzer with additional information about parameters (a scenario), e.g. possible variable ranges or values, allows reducing WCET over-estimation, either by improving the estimate, or by validating the initial estimate for a specific configuration or mode of execution. Nevertheless, exhaustively specifying constraints on all parameters is usually infeasible and identifying relevant ones (i.e. those impacting the WCET) is difficult. To address this issue, we propose the branching statement analysis, which uses a source-based heuristic to compute branch weights and that aims at listing unbalanced conditionals that correspond to system parameters. The goal is to help system-experts identify and formulate concise scenarios about modes or configurations that have a positive impact on the quality of the WCET analysis.

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

Keywords and phrases WCET Accuracy, Modes and Configuration, Flow Facts, Scenario Specification

Digital Object Identifier 10.4230/OASIcs.WCET.2014.93

1 Introduction

At Continental Automotive France SAS and in the automotive industry in general, the reuse of software is a major source of quality improvement and development cost reduction. Such reuse is enabled by product-platform approaches, for example by offering a clear decoupling between generic function development and application project integration. Therefore, the corresponding software modules are firstly developed in a platform context which makes them applicable in a large diversity of contexts, such as engine cylinder number, combustion type or fuel type. Abstraction of the system configuration and the hardware platform results in a configurable software module solution, for which the worst-case execution time (WCET) depends on a series of parameters. Additionally, software functions must be frequently adapted to changes of the configuration or enhanced by new features. Besides scheduling analysis and timing constraint verification, WCET analysis is applied in various phases of the development cycle of such an industrial project. In the design phase, it is used to estimate the computation power required by a brand new functionality in order to properly size the

* This work is supported by ANR W-SEPT.

© Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot;
licensed under Creative Commons License CC-BY
14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 93–102

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

hardware configuration. During module development, WCET analysis is applied to verify the compliance to certain platform rules, like, e.g., the maximum interrupt blocking time.

Due to a huge amount of unspecified settings in such a configurable environment the pessimistic assumptions of WCET analysis usually leads to a high over-estimation of the WCET, especially when it is applied in an early development phase.

Supplying precise information about possible values of relevant parameters can thus improve the quality of the WCET estimate or establish that the WCET estimate for the particular configuration coincides with the reported unconstrained WCET estimate. Manually identifying and specifying constraints on all relevant parameters is a tedious task. In order to assist such a task we propose an approach, *branching statement analysis*, that focuses on identifying crucial branching choices at control flow level. To this end, we identify parameters by analyzing and listing conditionals that are deemed WCET-relevant by our analysis. Then, a system-expert provides additional information about the particular configuration in form of input constraints (scenario). WCET analysis of the system under the supplied scenario either leads to an improvement of the WCET estimate for the analyzed mode, or validates the accuracy of the (global) analysis for this particular configuration.

The contribution of this work is an approach that makes the task of identifying relevant parameters in a configurable system less tedious, while gaining on the precision of the WCET estimate reported for the configuration. Our method relies on a static analysis tool to compute loop bounds and infeasible path information and can be applied iteratively to incorporate and refine value specifications of parameters. By relying on a source based timing heuristic, we identify *unbalanced* conditionals and guide the system-expert when constructing and/or refining short and precise scenario specifications about relevant parameters.

The rest of the paper is structured as follows. An overview of the approach is presented in Sect. 2, followed by an industrial point of view on WCET analysis in Sect. 3. A detailed description of our method is presented in Sect. 4, while Sect. 5 overviews effects of scenarios on the WCET analysis precision. We report on experiments in Sect. 6 followed by an overview of related work, Sect. 7, and we conclude in Sect. 8.

2 Motivating Example

The diagram shown in Figure 1 outlines the setup of our approach. Traditionally, to compute a WCET estimate, a (possibly empty) scenario and the program source code are supplied to a control flow analyzer (`oRange` [7]) that computes flow facts about the program (FFX [10]). The program binary and the flow facts are supplied as input to the low-level analyzer (`OTAWA` [1]) that computes a WCET estimate for the program. We extended the flow fact computation step by a Δ-computation step. The goal is to find unbalanced conditionals (in terms of weight, currently a syntactical measure) related to parameter values, such that a system-expert can focus on specifying parameters that are deemed relevant by the branching statement analysis. Specifying data constraints for those parameters yields a refined scenario that can again be supplied to the WCET computation step. The analysis can be re-run iteratively using the refined scenario.

As stated in Section 1, the WCET estimate usually depends upon a large number of parameters imposed by outside constraints. Therefore, one often is interested in estimating the WCET of the program in a specific mode of execution or configuration.

In this paper, we denote as the *configuration* of the system the hardware and software environment of a component. A *mode of execution* describes running the program under certain assumptions about the environment. *Parameters* then denote variables that reflect

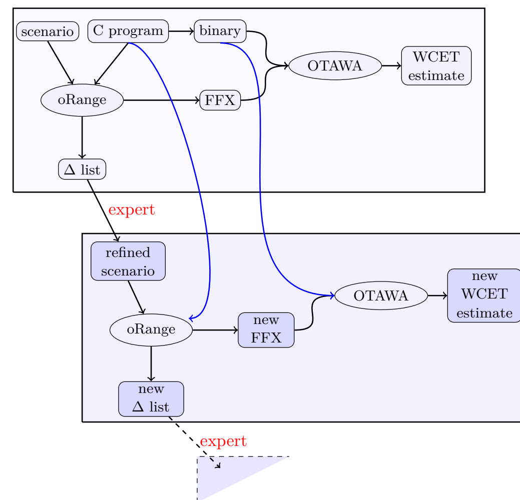


Figure 1 Introducing Δ-values to support scenario specification and refinement.

```
#include "missing.h"
int expensive() { /* ... */ }
int cheap() { /* ... */ }
int main () {
  for (int i = 0; i < 100; i++)
    if (max_speed > 250)
        expensive();
    else cheap();
}
```

Figure 2 A simple example.

```
Computing the balance information for the main function
Estimated cost of the function: 70804
1 accessible conditional statements
Delta 65000 at rex.c:22 in main (total count=100):
                       then=704; else=54;  // max_speed > 250
```

Figure 3 Analysis output: Δ-conditions for the example.

the configuration and/or the mode of the system, and by a *scenario* we denote a set of constraints on these parameters.

As an example, consider Figure 2 and suppose that `max_speed` is a variable among many parameters. Thus, the unbalanced conditional depends on variable values (parameters, configuration) that are not specified in the analyzed code. A system-expert knows that in a particular mode of execution or configuration (critical mode, model of vehicle, calibration) `max_speed` is less than 250. Restricting possible executions of the program by providing information about `max_speed` can therefore improve the WCET estimate for the configuration. Nevertheless, such a parameter first needs to be identified as relevant.

Figure 3 lists the Δ-conditions for the example, with a weight of 704 for the then-branch (Δ-weight of `expensive`) and a weight of 54 for the else-branch (Δ-weight of `cheap`). The conditional branches are executed within a loop, thus their weight needs to be scaled accordingly, resulting in a Δ-value of 65,000 over all loop iterations. As there is only a single `if`-statement present in the source the number of Δ-conditions is 1. The high Δ-value

indicates that supplying additional annotations about variables involved in the condition have a high impact on the WCET estimate computed for the function. Therefore, our approach proposes this variable as a relevant parameter.

Supplying input annotations for those parameters helps determine the program paths that are valid in the current scenario. For example, assuming a system where `max_speed` is known to be < 200 reduces the WCET estimate from 250,033 to 29,933 cycles.

3 The Need for Concise Scenario Specifications

WCET analysis in an industrial context is applied with different aims in a number of development phases. A regular WCET estimate is often not enough, especially when the lack of context makes it highly imprecise. We overview these phases to illustrate where the precision of WCET analysis can be improved by incorporating system-expert supplied scenarios.

First, the WCET of a piece of code (e.g. a module or a function) is estimated in isolation of any influence from other modules, be it effects from input or output channels, relations with other modules, variable interdependencies, interrupts or the system configuration. These estimations happen soon in the development cycle, by the function developer. Second, in a module aggregation phase the focus shifts to the aggregation of a number of modules (10 to 40) to form a package suited to be reused in application projects. In this phase, the code is an assembly of modules and sequences and no more modified.

The tuning of these settings results in exclusive behaviours and can be expressed as limitations on valid program paths. Therefore, in this phase, information about top level system integration and other high level system information can be supplied to a WCET analyzer in order to gain a more realistic view on the WCET on the package level. Finally, an application project view point on the integration phase that combines a number of, e.g. 70 to 100, aggregates leads to a WCET estimate used for scheduling analysis, as well for a proper configuration of the task set. At this point, the top level system configuration is set-up, and execution modes can be defined (including assumptions about engine speed, coolant temperature, etc.). Specifying such assumptions about a mode in a scenario allows to infer a tight contextual WCET estimate for the particular mode and configuration.

In a complete automotive system there are up to 40,000 variables, some of them tightly coupled due to system state inter-dependency and closed loop effect. Scenarios are necessary in order to model influences on the system. The tool we propose helps the system-expert identify parameters that are likely to impact the WCET estimate and should thus be specified in the scenario. This way, the WCET analysis of a module during the aggregation phase, or of an aggregate during the project integration phase can be provide a value closer to the WCET in real setting, while only specifying a low number of parameters. Scenarios therefore include information on parameters such as:

- input and internal variables, corresponding to system states, values of acquisitions, information coming from the network or diagnosis data. For example, the set of active functions is influenced by the state of the engine system (full load, idle speed).
- configuration parameters, influencing arrays sizes, loop bounds, and (de)activate code branches in runnables, runnables, or even full aggregates. They can be
  - configured statically, and must be assumed during validation. Those are usually concerned with “high-level” configuration such as the number of cylinders, cylinder banks, the type of sensors or the type of combustion.
  - configured dynamically (calibration parameters) that can be modified later by tuning engineers and influence computations, like e.g. interpolation- and index-tables.

4 Branching Statements Analysis

The list of conditional statements shown in Figure 3 is computed by our *branching statement analysis*. For each conditional statements of a C program the analysis computes a Δ-value, i.e. an indicator of how unbalanced its branches are in terms of weights, (currently) a source based heuristic. Thus, the analysis requires no binary program nor architecture information. We implemented this analysis on top of the control flow analyser `oRange`.

Analysis Input. The analysis takes as input a C program, consisting of C source and header files, together with an entry point and optional input-annotations. User-supplied annotations may contain information about the program data and the program control flow, that might not be inferable from the program code.

Analysis Output. The analysis outputs a *list of branching statements* of the program. Each branching statement is accompanied by:

- information to localize it in the source code;
- an upper bound on the number of executions, N;
- a list of its valid branches, together with their branch-weights, w_i;
- its Δ-value $= N \times \max_{i,j}(w_i - w_j)$; the formula reduces to $N \times |w_{\text{then}} - w_{\text{else}}|$, respectively $N \times w_{\text{then}}$, for `if-then-else`, respectively `if-then`, statements.

The list is sorted by Δ-values in decreasing order and outputs the weight of the program.

Control Flow Pre-Analysis. We assume that the branching statements analysis is preceded by a control flow analysis that computes the following information:

- Loop bounds for each loop of the program
- Branch validity for each branching statement (`if`, `switch`)

This information is deduced from the code and/or supplied in the form of input-annotations and is used in the weight computation.

Abstract Syntax Trees. The C program is represented as a collection of functions, each defined by a name and an *Abstract Syntax Tree* (AST). An AST is composed of statements, structuring sub-statements and expressions, and expressions composing sub-expressions and function calls. Figure 4 depicts an example of an AST.

In absence of recursion, linking the function names to their corresponding AST yields a directed acyclic graph (DAG) which root is the entry point.

Branching Statement Analysis. Attaches the following information to each node:

- its weight, which is a pessimistic numerical indicator of the execution time of the function, statement or expression
- a set of conditional statements possibly evaluated during its execution, together with upper bounds on the number of times they are evaluated.

The information attached to a node is computed from the information carried by its children. In the example shown in Figure 4, assuming that the control flow analysis infers that both branches are valid, the weight of the IF statement is defined by (1) $w_e + \max(w_{\text{S1}}, w_{\text{S2}})$. The execution count of conditional statements in e, S1 or S2 are combined applying formula (1). An entry in the list of Δ-conditions is added for each conditional statement. Additionally, it lists the execution count and branches S1 and S2 together with their respective weight.

Information is computed in a bottom-up manner from (valid) leaves to the root. We rely on a heuristic assigning integer weights to nodes representing elementary program operations, e.g. numerical operations or memory accesses. Weights and counts are updated taking into account inferred or supplied loop bounds, while branches marked as invalid do not appear in the output. Branching statement with only one branch are not added to the list.

Finally, the root node carries the weight of the whole program and is output together with the set of Δ-conditions of the program.

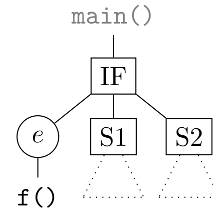


Figure 4 Example of Abstract Syntax Tree.

Limitations. The significance of the Δ-values depends on the constants and formulas used in the weight computation. Currently, and in our experiments, we use integer constants for different kinds of AST nodes. Nevertheless, a more complex scheme or using values computed by a static WCET analyzer is feasible. Statements which break the regular control flow do not receive a special treatment. As a consequence, a branch like `if (x) break;` inherently carries a low Δ-value.

5 Exploiting Scenario Specifications

An expert-supplied scenario, constructed by choosing and specifying values for suggested parameters, is likely to influence the computed WCET estimate. The control flow analyzer deduces path infeasibility, i.e. invalid execution under the current assumptions, from the additional annotations. Effects propagate to the low-level analyzer, as illustrated by the following example (Figure 5), as well.

In the following, we apply a simple cost heuristic for terminal nodes (cost 0 for numeric constants and field accesses and 1 for all other nodes) to identify scenarios and then use `oRange` and `OTAWA` to infer execution time estimates for the scenario.

Initially, the example is analyzed without scenario, finding all paths "valid". A follow-up low-level analysis computes possible addresses for the pointer access, resulting in ⊤ (no information) after the analysis of the `if`-statement. The assignment to `*p` results in a cache miss assumed by the low-level analysis.

Supplying value information (Figure 6) infers the branch invalid in the scenario and therefore marks the corresponding edge as `not-executed`. The low-level analyzer can thus ignore one of the branch for its analyses and infers an address for the access of `*p`, decreasing the original estimate, 95 cycles, to 83 cycles.

Before reporting on our experiments, let us summarize possible effects of utilizing Δ-conditions to specify concise but relevant scenarios:

The WCET path and estimate change when eliminating a branch on the WCET path. If a light branch was eliminated, the change is due to increased analysis precision of the following analyses. For example, when improved cache analysis results allow to infer block execution times (for previously selected blocks) that are below the execution time of alternative blocks (that were previously not selected).

The path is unchanged but the estimate is improved when a light branch is eliminated and the improvement of later analyses does not reduce block execution times of selected blocks below the execution time of their alternative blocks.

The WCET path changes while the estimate does not improve, in cases when a branch is eliminated but there exists another execution exhibiting a similar WCET estimate.

```
#include "missing.h"
int main ()
{
  int a, b;
  int *p;
  if (ext > 0)
    p = &a;
  else
    p = &b;
  *p = i;
}
```

Figure 5 Cache analysis fails to infer the address of `*p`.

```
// missing.h:
// no scenario, no value value
    ↪information
int ext;
(a)

// missing.h:
// scenario, additional value
    ↪information
int ext = 0;
(b)
```

Figure 6 "Invalidating" a path under a scenario arrows to infer the address of `*p`.

Finally, both the WCET path and the estimate are unchanged when eliminating a light branch and the improvement of following analyses does not propagate to blocks on the WCET path.

6 Experiments

Industrial Use-Case. The use-case is a 700 line C module provided by Continental Automotive France SAS. A system-expert manually identified and provided a list of 85 parameters and a scenario specification consisting of 30 parameter initializations. Branching statement analysis on the module reports 54 Δ-conditions, with Δ-values from 0 to 2034.

20 of the 30 parameters initialized in the provided scenario appear in the list of Δ-conditions, 18 of them exhibiting the highest 10 Δ-values of the list. 19 of the 54 Δ-conditions have low Δ-values (218 and less than 11) and no correspondence to the parameters in the scenario. As we rely on the parameter names to appear as operands in the Δ-conditions, a parameter may be linked to several Δ-conditions and vice versa.

Table 1 shows the result of WCET analysis of the module: column 1 lists the provided scenario, column 2 lists the number of specified parameters in the scenario and column 3 to 6 list the WCET estimate and improvement compared to the global WCET for an ARM7 lpc2138 platform, without and with a 1KB direct mapped data cache.

WCET analysis of the module without scenario, (1) global, reports 2553 (6883) as WCET estimate. WCET analysis of the expert-provided scenario, specifying 30 parameters, (2) full scenario, yields an improvement of 5%. Rows, (3)-(6), list the estimate and gain when specifying only those parameters involved in the i highest valued Δ-conditions.

To investigate the correspondence between high Δ-values and gain in the WCET estimate we inverted the scenario that initializes 3 parameters, row (3), thus forcing execution of the light branches of the corresponding conditionals, in row (7). To validate that specifications for parameters not contained in the list of Δ-conditions have little impact on the estimate, we supply the 10 parameter initializations that do not appear in any Δ-conditions, row (8).

Summarizing, branching statement analysis identified 20 of 80 parameters as important due to their high Δ-values in the list and they coincide with specified values in the expert-provided scenario. 10 parameters specified in the expert-provided scenario do not appear in the Δ-condition list and have almost no impact on the WCET estimate, while specifying only parameters identified in the 10 highest Δ-conditions still improves the estimate.

The experiment shows that our branching statement analysis can help system-experts focus on the relevant parameters from the vast number of possible parameters.

Table 1 WCET computation depending on parameters provided in scenarios

scenario	# parameters	no cache		cache	
(1) global, no scenario	0	2553	gain	6883	gain
(2) full scenario	30	2426	5%	6486	5.7%
(3) 3 highest Δ	3	2553	0%	6833	0%
(4) 8 highest Δ	10	2479	3%	6679	3%
(5) 9 highest Δ	14	2463	3.5%	6623	3.8%
(6) 10 highest Δ	18	2448	4%	6568	4.6%
(7) inverted 3 highest Δ	3 (inverted)	2055	19%	5795	15.8%
(8) none of Δ	10	2551	0.08%	6831	0.03%

Table 2 Potential for WCET improvement for Mälardalen benchmarks.

program	**# Δ**	**highest Δ**	**overall weight**	**ratio**
sqrt	4	4458	4540	98.19
expint	3	35800	38026	94.14
prime	4	12,773,225	86,858,003	14.70
crc	5	6144	49223	10.45
ndes	5	384	43930	0.8
st	4	743	114,137	0.65
fir	2	30	204,371,956	0.00001

Branching Statements Analysis of Mälardalen. Even though our approach is motivated by industrial need, branching statement analysis provides relevant results when applied to the Mälardalen benchmark suite [6]. The benchmarks lack scenarios, therefore we target identifying interesting variables and program points instead of system parameters.

Column # Δ in Table 2 states the number of unbalanced conditionals found in the program. Highest Δ lists the highest Δ-value reported. The last two columns list the weight of the program and its ratio to the highest Δ-value. The ratio is an indication of potential improvement when supplying additional information for the conditional is feasible.

In programs like `expint` or `sqrt`, the highest Δ-value weighs over 94% of the total weight, which hints at a relevant program point[1]. There is potential for improvement in programs `crc` and `prime`, yet they lack the opportunity to specify annotations. The low Δ-values of the rest of the programs suggest a low potential for improvements.

As expected, the nature of the benchmarks, lacking system parameters, prohibits scenario specification, but interesting program points can still be identified by the analysis.

7 Related Work

There is a body of work that is related to branching statement analysis.

The weight computation and propagation in the AST is comparable to tree-based WCET computation, whereas identifying relevant conditionals is related to (static) profiling. In contrast to the other approaches, it aims at helping system-experts identify relevant parameters

[1] In `expint` the high weight of the conditional is due to an inner loop with a high execution count, while it is run only once. In `sqrt` the condition guards a light special case of the computation.

by identifying unbalanced conditionals on source level using a simple heuristic for weights.

Tree-based WCET computation can be applied as alternative to IPET [3]. Estimates are computed by a bottom-up traversal of the parse tree of a program. Leaves represent basic blocks and are annotated with timing information. The program is traversed and for each node timing information is computed from the timing information of its child nodes. In contrast to tree-based WCET computation, branching statement analysis does not rely on timing information of basic blocks but uses a heuristic to compute timing information from the syntactic expressions.

In [2] the authors present the *criticality metric* to express for each statement how important it is for the global WCET. This allows finding out, for a piece of code, how close the WCET of paths passing through the piece of code is to the global WCET. The ratio between Δ-values and the weight of the function indicates a potential for improving the WCET estimate by supplying a scenario.

Dynamic program profiling usually executes an instrumented program in order to explore the performance and/or flow [5]. Static approaches generate static profiles, e.g. by computing probabilities for decisions at branching points [9]. A major problem in profiling is to find input values that capture the performance profile of the application. Our analysis does not execute or instrument the program and helps to identify relevant parameters.

The author of [8] sketches the ingredients for a static profiler. The output is an unfolded inter-procedural control flow graph computed from the results of multiple static analyzers that identify feasible paths and compute loop bounds. The graph is guaranteed to include the worst-case behaviour of the program. Static analysis results are used similarly in branching statement analysis but instead of execution frequencies it computes weights.

Most closely related is a work on scenario detection [4], following a comparable approach, with a slightly different goal. The influence coefficient is computed for parameter variables, which are identified as variables or fields that are assigned once and do not change over the program execution. Scenarios are constructed such that they split and cover the domain of the parameters, allowing to WCET analyze each scenario. The maximum execution time among the scenarios is then considered as the WCET of the application. In contrast to [4], our notion of parameters allows for changes in value, and they might not be assigned in the code at all. Instead of discovering variables, we match conditions with high Δ-values to a number of pre-selected parameters. Furthermore, our weight heuristic should be independent of both the underlying architecture and WCET analysis, in order to be able to apply it in different phases of development.

8 Conclusion and Outlook

We presented branching statement analysis, an approach to guide system-experts in generating concise, but relevant, scenarios for the WCET analysis of systems. Such systems are usually composed of a number of components, influenced and controlled by a vast number of parameters. The high number of parameters usually results in an overly pessimistic WCET estimate and prohibits exhaustive specification of scenarios that allow to obtain a more realistic estimate. Branching statement analysis allows to find unbalanced conditions that depend on parameters. Additional information about a low number of identified parameters can already significantly improve the WCET estimate or validate the estimate for the scenario. To this end, we see branching statement analysis as an important step towards a more functionally representative (or plausible) WCET, instead of a purely structural one.

We are currently applying part of our approach manually, i.e. we rely on scenarios specified

by system-experts. To further improve trust, such scenarios could be verified, whenever possible, by automated tools.

We plan to integrate our approach in a fully automatic setting, where additional information is inferred using static analysis. Branching statement analysis could be used to select program points where counter instrumentation and analysis is applied to infer constraints between instrumented basic blocks.

Additional effort will be put into a richer set of supported input-annotations in the scenario, currently restricted to information about variable values and boolean information about the execution of conditional branches. The system-expert might have at its disposal information like limits on the number of execution of a statement or contexts in which a statement must or must not be executed.

References

1. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.
2. Florian Brandner, Stefan Hepp, and Alexander Jordan. Static Profiling of the Worst-case in Real-time Programs. In *Proc. of RTNS*, pages 101–110, 2012.
3. Matthew Emerson, Sandeep Neema, and Janos Sztipanovits. *Handbook of Real-Time and Embedded Systems*, chapter 6. CRC Press, 2006. ISBN: 1584886781.
4. Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, and Henk Corporaal. Automatic Scenario Detection for Improved WCET Estimation. In *Proc. of DAC*, pages 101–104, 2005.
5. Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proc. of SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
6. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET*, pages 136–146, 2010.
7. 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 *Proc. of RTCSA*, Taiwan, 2008.
8. Adrian Prantl. Towards a Static Profiler. Technical report, Vienna University of Technology, 2009.
9. Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. In *Proc. of MICRO*, pages 1–11, 1994.
10. Jakob Zwirchmayr, Armelle Bonenfant, Marianne de Michiel, Hugues Cassé, Laura Kovacs, and Jens Knoop. FFX: A Portable WCET Annotation Language. In *Proc. of RTNS*, pages 91–100, 2012.

Towards Automated Generation of Time-Predictable Code*

Daniel Prokesch, Benedikt Huber, and Peter Puschner

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

Abstract

Knowledge of the worst-case execution time of software components is essential in safety-critical hard real-time systems. The analysis thereof is not trivial as the execution time depends on many factors, including the underlying hardware platform, the program structure, and the code produced by the compiler. Often, the execution time is variable and highly sensitive to the input data the program has to process. This paper presents a code transformation applicable in a compiler backend that produces time-predictable code. The resulting code contains a single input-data independent execution path, in order to obtain programs of stable timing behaviour. The transformation technique has been validated by applying it on a number of benchmarks. Experiments show a reduction of execution time variability, at acceptable costs for the single execution path.

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

Keywords and phrases Single-Path, Graph Transformation, Predictable Code, Compiler

Digital Object Identifier 10.4230/OASIcs.WCET.2014.103

1 Introduction

Hard real-time systems are characterised by the fact that their correctness does not only depend on the computational results but also on the timely delivery thereof – failing to provide a result in time will potentially have catastrophic consequences. For this reason, knowledge of the worst-case execution time (WCET) of a software component in the context of a hard real-time system is essential. Determining the WCET of a program is hard in general, and implies solving two sub-problems: modelling the timing behaviour of the hardware components and determining the possible program execution paths. On the software side, components tend to be highly complex with regard to their behaviour in the presence of different input data and their context. This makes precise automatic analysis of the possible program execution paths intractable in general, as the number of paths grows exponentially in the number of control flow alternatives. These practices are contradictory to at least two key principles of hard real-time systems design: simple structures and composability.

One proposed solution to get around the complexity of WCET analysis is the *single-path approach* [4]. A single-path program is characterised by the fact that it has a singleton program execution path, which makes path analysis superfluous and reduces the need for complex timing models. The execution time of the program is stable with respect to varying

* This work was partially funded by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST) and the EU COST Action IC1202: Timing Analysis on Code Level (TACLe).

© Daniel Prokesch, Benedikt Huber, and Peter Puschner;
licensed under Creative Commons License CC-BY

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 103–112

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

```
     cond := ...
     if (!cond) goto Lelse
Lthen:
     x := a + 1
     goto Lend
Lelse:
     x := b - 2
Lend:
     ...
```

(a) Update of `x` by branching code.

```
          cond := ...
( cond) x := a + 1
(!cond) x := b - 2
          ...
```

(b) Update of `x` by predicated assignments.

Figure 1 Whereas 1a has two alternative execution paths, each containing a different assignment of `x`, 1b consist of a single linear sequence of instructions containing both assignments of `x` with disjoint predicates, such that only one of the two assignments has an effect.

input data, making its temporal behaviour predictable. Ideally, the WCET of a single-path program should be obtainable by simple measurement of one execution.

A set of rules required to transform a piece of code given in a high-level representation to a single-path version was presented in [5]. These rules express the translation from a high-level, well structured source language to predicated statements. While they provide a clear conceptual understanding, they are not sufficient to implement the single-path conversion in a compiler backend. In modern state-of-the-art compilers, source languages are translated to a common intermediate representation (IR), on which optimisations are performed in a language-independent way. A code generator (compiler backend) then translates the optimised IR to target-specific machine language. Control flow is explicit in this representation and typically more general than what can be expressed by a well-structured programming language. This work describes the single-path transformation from a low-level perspective, as graph transformation technique on program control flow graphs, amenable for implementation in a compiler backend. This facilitates automated generation of predictable machine code from any piece of WCET-analysable code.

The rest of this paper is organised as follows: Section 2 provides the theoretical foundations of the single-path approach and basic definitions required for the transformation technique. Section 3 describes the transformation technique with the corresponding execution semantics, to transform almost arbitrary control flow graphs to graphs yielding a single execution path. Section 4 documents the validation of the approach by experiments and the effect on the execution behaviour obtained from these experiments. A short overview of related work is given in Section 5, before we conclude with Section 6.

2 Background and Preliminaries

2.1 Predicated Execution

The single-path transformation is based on *predicated execution.* A predicated instruction is executed conditionally depending on the value of a Boolean predicate, referred to as *guard*: if the predicate value is true, the instruction is *enabled* and executes as expected, otherwise it is *disabled* and exposes the behaviour of a no-op, that is, the hardware state (register file, memory contents) is not altered as a result of the instruction. By means of predicated instructions it is possible to replace changes in control flow by conditional execution of an instruction sequence, like the code snippets in Figure 1 suggest. In both variants the contents of variable `x` are updated depending on the evaluation of `cond`, but by different means.

Predicated execution, as we intend to exploit it, requires certain assumptions about the target hardware. We assume that all instructions of our target instruction set are predicatable and that the instruction latencies are independent of the operand values, in particular for the predicate operand.

2.2 Basic Definitions

Before we detail on the transformation procedure, we briefly give some basic definitions.

A *basic block* (BB) is a straight-line sequence of instructions with one entry point and one exit point. A *control flow graph* (CFG) is a directed graph with basic blocks as nodes and models possible execution paths through a function. We require that a CFG has a distinguished entry node and a distinguished exit node. A node v with more than one successor is associated with a *branch condition*, $cond_v$, a Boolean condition that determines which successor to take on a path.

A node x *dominates* a node y (denoted as $x \operatorname{dom} y$) if every path from the start node to y must go through x. A node x *postdominates* a node y (denoted as $x \operatorname{pdom} y$) if every path from y to the exit node must go through x. x strictly (post-)dominates y if $x \operatorname{dom} y$ (resp. $x \operatorname{pdom} y$) and $x \neq y$. The immediate (post-)dominator x of a node y is the unique node that strictly (post-)dominates y but does not strictly (post-)dominate any other node that strictly (post-)dominates y. (Post-)dominator information commonly is presented in form of a *(post-)dominator tree* in which the entry (resp. exit) node is the root and each node (post-)dominates only its descendants in the tree, i.e., the parent node of each node is its immediate (post-)dominator.

A *loop* L is a strongly connected set of nodes in the flow graph. A *natural loop* has a distinguished entry node, the loop *header*, which dominates all nodes in the loop, and a *back edge* that enters the loop header. Given a back edge (n, d), the natural loop of the edge is defined as d plus the set of nodes that can reach n without going through d. n is also called *latch*. An edge (u, v) is an *exit edge* of loop L if $u \in L$ and $v \notin L$. In a *reducible CFG*, every cycle contains a back edge that can be associated with a natural loop. Unless two natural loops have the same header, they are either disjoint or one is nested within the other. If two natural loops share the same header, we treat them as a single loop identified by the header.[1] Furthermore, every node v has a unique header, denoted as $header(v)$, which is the header of the innermost loop it is contained in. We consider an entire procedure as pseudo-loop with the entry node as pseudo-header, such that every node of the CFG except the entry node has a header. Removing all back edges from a reducible CFG results in an acyclic flow graph, the *forward control flow graph* (FCFG).

Given a CFG $G = (V, E)$, nodes $u, v, x \in V$ and an edge $(u, x) \in E$, v is *control dependent* on u (or on edge (u, x)) if v postdominates x but not u [1]. For any node $v \in V$ the set of its control dependence edges is denoted by $CD(v)$:

$$CD(v) \equiv \{(u, x) \mid v \text{ is control dependent on } (u, x)\}$$

The control dependence function induces a partitioning on the nodes of the CFG into equivalence classes: If two nodes $v, w \in V$ are control dependent on the same set of edges, then on every path π in G from the entry to the exit node, v is on path π if and only if w is on π.

[1] This differs from the common practice that if a loop is a proper subset of another loop, the former is treated as inner loop and the latter as outer loop.

We require two properties of the CFG, which can be provided by suitable preprocessing: We assume that the CFG is reducible, and that every node in the CFG must have an outdegree of at most two.[2] The second requirement implies that we can name the *dual edge* of an edge if its source node has more than one successor: Let $u \in V$ be a node with two successors v, $w \in V$. Then, the *dual edge* of edge (u, v) is (u, w), and the dual edge of (u, w) is (u, v).

3 The Single-Path Graph Transformation

We describe our technique as a graph transformation from a source CFG to a target CFG that is extended by predicated execution semantics. In the beginning, we define admissible executions in the source CFG in the presence of loop bounds and present our model of predicated execution in a graph.

An *execution* of a control flow graph G is a path from the entry to the exit node. We require that every loop header in the CFG is associated with a number, the *(local) loop bound*, that limits the number of times the header is (re-)entered on a path before the corresponding loop is left via an exit edge.

▶ **Definition 1** (Admissible execution of a graph). An *admissible execution of a control flow graph* G is a path π in G on which each loop bound for any header $h \in \pi$ is respected.

For predicated execution, every node is associated with a guarding predicate that determines whether the node is enabled or disabled. Predicates can be seen as part of state which is altered as nodes are visited along a path. Following semantic actions can be performed by nodes and edges:

- A node may set a predicate to the branch condition or its negation. This action is predicated itself by the node's guard.
- An edge may set or clear a set of predicates (set them to true/false), unconditionally.
- An edge may copy the value of one predicate to another predicate, unconditionally.

▶ **Definition 2** (Single-Path Graph Transformation). Let $G = \langle V, E \rangle$ be a graph with local loop bounds. The *Single-Path Graph Transformation* obtains a graph G^{SP} extended by predicated execution, and a path π^{SP} in G^{SP}, such that for any admissible path π in G, the sequence of nodes along π equals the sequence of enabled nodes along π^{SP} in G^{SP}.

The single-path graph transformation computes a singleton path π^{SP} through G^{SP} that includes every admissible execution path in the source graph. In the following, we describe how we compute G^{SP} and π^{SP}. Section 3.1 reviews the computation of predicates and the conversion of acyclic graphs. In Section 3.2, these ideas are extended to obtain single-path loops, that is, loops that may contain nested loops and have a fixed iteration count. Section 3.3 summarises the construction of the single-path graph G^{SP} by composition of single-path loops.

3.1 From Control-Flow to Predicates

Our transformation technique is based on the *RK algorithm* of Park and Schlansker [3]. Their motivation is the ability to speed up the execution of innermost loops by means of a software

[2] In practice, this implies that jump tables (e.g. resulting from `switch` statements) must be replaced by cascades of two-way alternatives.

pipelining technique, which requires a linear sequence of instructions without control-flow changes. As the algorithm is applied to innermost loops only, the CFGs under consideration are acyclic in nature. We show how to apply the algorithm to transform an acyclic CFG to a linear sequence of predicated basic blocks.

Given an acyclic graph $G = (V, E)$, the algorithm assigns a unique predicate to each of the equivalence classes induced by the control dependence relation. The set $CD(v)$ for all $v \in V$ is computed by means of the postdominator tree (PDT) of G [1]: Each node $v \in V$ with two successors is identified. Then, for each successor w of v, the PDT is walked upward, starting at w, until (excluding) the immediate postdominator of v is reached. As each node u is visited during the walk, the edge $(v, w) \in E$ is added to the set $CD(u)$.

The set of predicates is denoted as P. Each predicate $p_i \in P$, $i \geq 0$ corresponds to a set of (control dependence) edges, which is expressed as function $K(p_i)$. The function $R(v)$ associates each $v \in V$ with a predicate $p_i \in P$ such that, for v, $w \in V$,

$$CD(v) = CD(w) \quad \Leftrightarrow \quad R(v) = R(w) = p_i \;\wedge\; K(p_i) = CD(v) = CD(w)$$

Control flow is mapped to predicate values in the resulting sequence of guarded blocks with the following goal: In any execution, for any $p_i \in P$, if a control dependence edge $e \in K(p_i)$ would have been taken in the original acyclic graph, p_i should be true, such that every node $v \in V$ with $R(v) = p_i$ is enabled, otherwise p_i should be false (and the corresponding nodes disabled). To this end, following steps need to be performed:

1. For each predicate $p_i \in P$ and each edge $(u, v) \in K(p_i)$, add a predicate assignment of the form $p_i \leftarrow cond_u$ as semantic action to node u, if v follows u when $cond_u$ is true, else add assignment $p_i \leftarrow \neg cond_u$ to u.
2. Guard nodes $v \in V$ by predicate $R(v)$.
3. As predicate assignments are potentially disabled in an execution, it is necessary to care for a correct initialisation of predicates. Therefore, predicates are initialised to false at an artificial entry edge as necessary.
4. The nodes are reconnected in the order of a topological sort, such that a linear sequence of predicated nodes is obtained.

3.2 Single-Path Loops

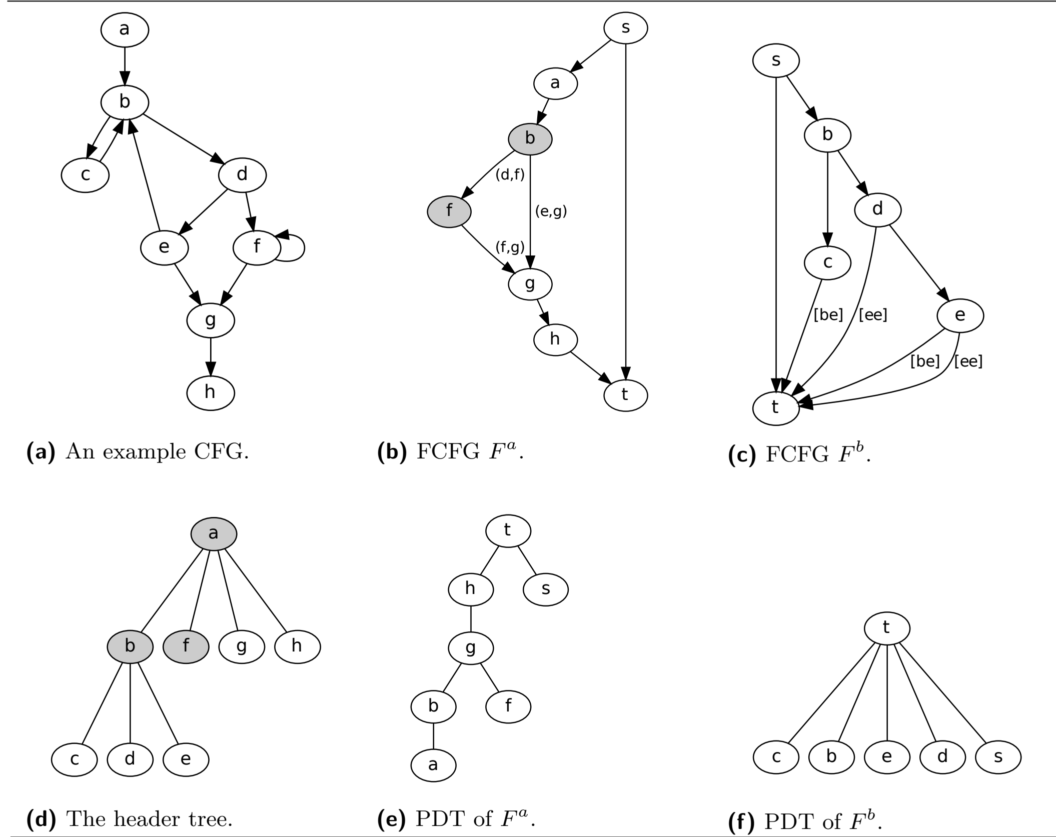
At the beginning of the transformation, we identify loops in the CFG and compute $header(v)$ for each $v \in V$, e.g, by following the procedure in [8].[3] For each loop, we identify back edges and exit edges.

We now consider a loop of the CFG in isolation. Let L^h be a loop of the CFG with header h.[4] We construct the acyclic FCFG induced by L^h, and augment it by two distinguished nodes s^h and t^h, and edges (s^h, h), (s^h, t^h), and (ℓ, t^h) for all latches ℓ of L^h, and (e, t^h) for all sources e of exit edges of L^h. Every node in the resulting graph, denoted as F^h, is dominated by s^h and postdominated by t^h.

A key insight for the extension of the procedure in Section 3.1 is that loops of a reducible flow graph can be represented compacted into a single node. Consequently, each contained inner loop is compacted into a single node in F^h, namely its header. As a result, the outgoing edges of an inner loop header correspond to the exit edges of the inner loop.

[3] $header(v)$ is the node into which v is eventually contracted during reduction of the CFG (procedure REDUCE in [8]).

[4] In the following, we use the header h in superscript to denote information specific to a single loop identified by h, to avoid ambiguities.



(a) An example CFG. **(b)** FCFG F^a. **(c)** FCFG F^b.

(d) The header tree. **(e)** PDT of F^a. **(f)** PDT of F^b.

Figure 2 An example to illustrate the single-path graph transformation steps.

▶ **Example 3.** The transformation is best illustrated by an example (Figure 2). Figure 2a shows the CFG, with entry node a and exit node h. Figure 2d depicts the header information for each $v \in V$ as tree: the parent of node v is $header(v)$, and each header is drawn shaded. Apart from the outermost pseudo-loop L^a with header a, the CFG contains two loops, loop L^b with header b and the self-loop L^f consisting of node f. Figure 2b depicts the FCFG for L^a, where b and f represent inner loops L^b and L^f compacted into a single node, and the outgoing edges are labelled with the corresponding exit edges. Figure 2c depicts the FCFGs for L^b, where the back edges and exit edges are represented by the edges labelled with [be] and [ee], respectively. The FCFG F^f for the self-loop L^f is not shown.

Next, $CD^h(v)$ is computed for all $v \in F^h$, with the aid of the postdominator tree of F^h, as described in Section 3.1. The main difference lies in the inclusion of control dependence edges: Edges originating from inner loop headers in F^h correspond to exit edges of the inner loops, and these exit edges have to be added to the respective control dependence sets.

Following the description in Section 3.1, R^h and K^h are computed from CD^h, nodes are guarded according to R^h and predicate assignments are added according to K^h. R^h maps predicates $p_i^h \in P$ to each node $v \in F^h \setminus \{s^h, t^h\}$. By convention, let $R^h(h) = p_0^h$. If v represents an inner loop, $R^h(v)$ is not the final guard of the inner loop header, but a predicate whose value decides whether the inner loop is enabled at least once. Note that an edge $(u, v) \in K^h(p_i^h)$ may be an exit edge originating from an inner loop, and hence node u is not necessarily a node in F^h. For edge (s^h, h), we add the semantic action to clear all predicates $p_i^h \in P$ for $i > 0$, to provide a correct initialisation of predicates.

In the acyclic graph F^h, the header h and every node that postdominates h are only control dependent on the edge (s^h, h), which corresponds to the loop entry. If the header predicate $R^h(h) = p_0^h$ is true, all associated guards of the nodes on any path in F^h from s^h to t^h are set to true as a result of the semantic actions, while the guards of the nodes that are not on that path remain false. In the transformed single-path graph, in any execution, all nodes of $F^h \setminus \{s^h, t^h\}$ are visited in the order of a topological sort.

So far, we have dealt with a path through a single iteration of the loop. In the source CFG, except for the top-level pseudo loop, a loop L^h is entered via its header h up to N times on an admissible path, where N denotes the local loop bound of L^h. As we want to obtain a single execution path that contains all admissible paths through the source CFG, this path must contain the sequence of nodes in L^h at least N times. Therefore, the single-path graph G^{SP} will contain a back edge from the last node in the sequence of nodes of L^h to the header h, which is taken exactly $N-1$ times in the single execution path π^{SP}. The semantic action of this edge is, like on (s^h, h), to clear all predicates $p_i^h \in P$ for $i > 0$.

The semantic actions on predicates need to be extended, otherwise the header predicate p_0^h is never set to false. If p_0^h is false at the beginning of an iteration, no guard from a loop member ever becomes true, and all nodes of L^h remain disabled in that iteration. Hence, on the single execution path, p_0^h must be false starting with the $(i+1)$-th up to the N-th iteration, if on an admissible path in the source graph the loop iterates i times. Recall that a loop is left via one of its exit edges. As each exit edge necessarily has a dual edge (otherwise, the loop header would not be reachable from the edge source, disqualifying the latter as loop member), we add the dual edge of each exit edge of L^h to $K^h(p_0^h)$.

▶ **Example 4.** Figure 2e depicts the PDT for the computation of CD^a in FCFG F^a. $CD^a(x) = \{(s, a)\}, \forall x \in \{a, b, g, h\}$, and $CD^a(f) = \{(d, f)\}$. Note that (d, f) is the exit edge of L^b that corresponds to (b, f) in F^a. Consequently, we assign $R^a(x) = p_0^a, \forall x \in \{a, b, g, h\}$ with $K^a(p_0^a) = \{(s, a)\}$, and $R^a(f) = p_1^a$ with $K^a(p_1^a) = \{(d, f)\}$. A topological order of the nodes is $\langle a, b, f, g, h\rangle$.

Figure 2f shows the PDT for F^b. $CD^b(b) = \{(s, b)\}$, $CD^b(c) = \{(b, c)\}$, $CD^b(d) = \{(b, d)\}$, and $CD^b(e) = \{(d, e)\}$. We assign $R^b(b) = p_0^b$, $R^b(d) = p_1^b$, $R^b(e) = p_2^b$, $R^b(c) = p_3^b$. Obtaining the respective $K^b(p_i^b)$ is straightforward, but we have to extend $K^b(p_0^b)$ by the dual edges $\{(e, b), (d, e)\}$ of the exit edges of L^b. A topological order of the nodes is $\langle b, c, d, e\rangle$.

For the single node self loop f, $R^f(f) = p_0^f$ with $K^f(p_0^f) = \{(s, f), (f, f)\}$.

3.3 Composition of the Single-Path Graph

We construct the single-path graph $G^{SP} = (V, E^{SP})$ by a preorder traversal of the loop header tree, visiting the nodes on each level in a topological sort order, and adding edges for G^{SP} as required. The recursive construction algorithm is sketched in Figure 3. It is invoked with the entry node of the source CFG.

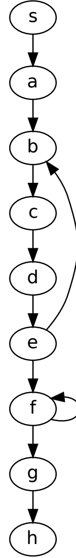
When a loop header h is visited, we construct F^h, compute R^h, K^h, guard nodes and add semantic actions, as described in Section 3.2. On the entry edge to an inner loop $L^{h'}$, we add the semantic actions of $(s^{h'}, h')$ (clearing the predicates). Additionally, we add another semantic action to copy the predicate of h' in the outer loop to the predicate of h' in the inner loop. For the CFG, an artificial entry edge is introduced to initialise all predicates of the top-level pseudo loop to false and the predicate of the entry node to true.

For our example, the resulting single-path graph is depicted in Figure 4, together with the guards and semantic actions.

Function COMPOSESP(h: header node)

1. Compute R^h, K^h
2. Add predicate assignments according to K^h
3. Guard header node h by $R^h(h)$ $(= p_0^h)$
4. $last = h$ // keep a reference to the last node visited
5. **for each** $n \in F^h \setminus \{h, s^h, t^h\}$ in topological sort order :
6. $E^{SP} = E^{SP} \cup \{(last, n)\}$
7.   **if** n is a loop header :
8.   Add semantic actions to entry edge $(last, n)$: $p_0^n \leftarrow R^h(n)$; $\forall i > 0 : p_i^n \leftarrow 0$
9. $last =$ COMPOSESP(n) // process inner loop
10. $E^{SP} = E^{SP} \cup \{(last, n)\}$ // add back edge to inner loop
11.   Add semantic actions to back edge $(last, n)$ of inner loop: $\forall i > 0 : p_i^n \leftarrow 0$
12.   **else** :
13.   Guard node n by $R^h(n)$
14. $last = n$
15. **return** $last$

Figure 3 Algorithm for the composition of the single-path graph.



Node/Edge	Guard	Semantic action
a	p_0^a	–
b	p_0^b	$p_1^b \leftarrow cond_b$; $p_3^b \leftarrow \neg cond_b$
c	p_3^b	–
d	p_1^b	$p_0^b, p_2^b \leftarrow \neg cond_d$; $p_1^a \leftarrow cond_d$
e	p_2^b	$p_0^b \leftarrow cond_e$
f	p_0^f	$p_0^f \leftarrow cond_f$
g	p_0^a	–
h	p_0^a	–
(s, a)	–	$p_0^a \leftarrow 1$; $p_1^a \leftarrow 0$
(a, b)	–	$p_0^b \leftarrow p_0^a$; $p_1^b, p_2^b, p_3^b \leftarrow 0$
(e, f)	–	$p_0^f \leftarrow p_1^a$

Figure 4 The complete single-path graph resulting from the transformation. For semantic actions, 0 is false, 1 is true, and $cond_v$ is the branch condition in node v.

4 Experiments

To validate the single-path graph transformation, we have created a simulation framework, which serves two purposes. First, it allows for an experimental validation of the transformation procedure, with arbitrary CFGs as input. Second, it provides means to evaluate the estimated execution cost of the single-path-transformed graph relative to the original CFG.

In the framework, after a given CFG G is transformed to G^{SP} and extended by semantic actions on predicates, admissible paths π through G are chosen by random. For one admissible path, the branch conditions are recorded. Then, the single execution path π^{SP} through G^{SP} is walked, and predicates are updated according to the recorded branch conditions in π. The

Table 1 Experiments on the Mälardalen benchmarks. Functions *adpcm/decode*, *fdct/fdct*, *jfdctint/jpeg_fdct_islow*, *loop3/main*, and *matmult/Test* are omitted because their generated CFGs already have a single execution path. Functions *adpcm/encode*, *bs/binary_search*, *bsort100/BubbleSort*, *cnt/Test*, *crc/icrc*, *duff/duffcopy*, and *edn/main* have almost a single execution path with a ratio below 1.10.

Benchmark/Function	Mean	Std.Dev.	Min	Max	SP	$\lvert P \rvert$	Ratio
adpcm/upzero	73.21	21.57	53	96	125	3	1.30
compress/compress	1178.13	840.65	451	3589	4200	28	1.17
cover/swi120	1512.05	18.08	1475	1570	2655	7	1.69
expint/expint	1940.28	18.45	1892	1984	2736	4	1.38
fir/fir_filter_int	1704.76	451.32	595	2618	3236	6	1.24
insertsort/main	496.24	109.97	246	750	832	6	1.11
janne_complex/complex	695.98	181.55	215	1135	1381	6	1.22
lcdnum/num_to_lcd	33.94	0.98	30	36	190	45	5.28
lms/main	94794.29	4106.37	86184	103839	150209	32	1.45
ludcmp/ludcmp	301329.73	16345.35	275478	328187	415544	16	1.27
minmax/main	55.21	7.18	48	70	83	7	1.19
minver/minver	135115.74	55424.42	70709	197066	371563	20	1.89
qsort-exam/sort	4609.72	1128.15	908	7253	12004	22	1.66
qurt/qurt	590.77	686.31	44	1838	2208	15	1.20
select/select	4345.63	896.34	2043	7196	11219	16	1.56
statemate/FH_DU	232.39	39.56	157	305	360	25	1.18

sequence of enabled nodes in π^{SP} is compared against the sequence of nodes of π and must be identical.

Furthermore, we generated the CFGs from functions from the well-established WCET benchmarks of the Mälardalen WCET research group[5], providing a simple cost model and local loop bounds.[6] We recorded the execution cost for 100 randomly generated paths through each CFG, and obtained the mean cost, the standard deviation, and the minimum and maximum observed cost. We computed the ratio of the cost of the single-path execution (SP) to the maximum observed cost (Max). As we are interested in the worst case, we forced the paths chosen in G to always execute the maximum number of loop iterations. In addition, we listed the required number of predicates ($\lvert P \rvert$). The results of this comparison are shown in Table 1. The ratio SP/Max was mostly below 1.9. In one case (lcdnum/num_to_lcd), the ratio is about 5.4, which stems from the fact that the function contains a switch statement that is serialised in the single-path graph.

5 Related Work

Techniques other than the single-path approach have been proposed to make code more predictable for WCET-analysis. Apart from avoiding problematic code constructs in the first place (e.g. indirect calls, irreducible and input-data dependent loops, recursion), code transformations have been suggested to reduce the number of paths required to be analysed by making infeasible paths explicit or factor out code blocks with constant execution time [2]. Both transformations seem hard to be performed automatically in a compiler and no general

[5] Accessible online: `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

[6] The cost of executing a block is the number of instructions in the block. We used local loop bounds bounds recorded by simulation, and for the loops not executed in the simulation, we added a bound of 20.

solution has been supplied so far. Compiler support to aid WCET analysis by providing information available during compilation to the timing analysis is an orthogonal approach to obtain more predictable code [6].

6 Discussion and Outlook

We have presented the single-path graph transformation, a technique to transform any reducible control-flow graph into a graph with a single execution path of predicated nodes. The goals of the single-path approach are to minimise control flow complexity and to obtain code with stable, input-data independent timing behaviour. The here-presented technique can be implemented as part of a compiler backend to generate time predictable code in an automated way.

Because the single-path approach serialises control flow, it has been criticised to be too costly to be applied in practice. In our experiments, the costs stay within reasonable limits (the ratio is below 1.9 in all but one cases) when the worst case is considered, for a simple model. Furthermore, the cost could be compensated partially by means of hardware support particularly suitable for single-path code, e.g., by providing a multiple-issue pipeline, instruction prefetching, or hardware loops.

We are implementing the single-path transformation as part of a compiler backend for the Patmos processor, a multi-core processor designed for high performance at high time-predictability [7], in the T-CREST project (`http://www.t-crest.org/`). A next logical step to improve the transformation would be to omit input-data independent regions during if-conversion, as they do not contribute to input-data induced variability.

References

1 Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

2 Hemendra Singh Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying wcet analysis by code transformations. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, September 2004.

3 Joseph C.H. Park and Mike Schlansker. On predicated execution. Technical report, Hewlett Peckard Software and Systems Laboratory, May 1991.

4 Peter Puschner. The single-path approach towards wcet-analysable software. In *2003 IEEE International Conference on Industrial Technology*, volume 2, pages 699–704 Vol.2, 2003.

5 Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In *Proc. SAFECOMP 2012 Workshops (LNCS 7613)*, pages 382–391. Springer, 2012.

6 Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, 2013.

7 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, March 2011.

8 Robert Endre Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, December 1974.