10th International Workshop on Worst-Case Execution Time Analysis

WCET 2010, July 6, 2010, Brussels, Belgium

Edited by

Björn Lisper

OASICS

Editor

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

ACM Classification 1998
C.4 Performance of Systems, D.2.4 Software/Program Verification, J.7 Computers in Other Systems

ISBN 978-3-939897-21-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
December, 2010.

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

Digital Object Identifier: 10.4230/OASIcs.WCET.2010.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.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Session 1: Cache and low-level analysis

Session 2: Measurement-based methods and flow analysis

Invited talk

Session 3: Parallel systems, model checking

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Session 4: Benchmarks, memory allocation

Preface

On July 6, 2010, the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010) was held in Brussels, Belgium. The workshop was organised as a satellite event of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10). The goal of this annual workshop is to bring together people from academia, tool vendors, and tool users in industry who are interested in all aspects of timing analysis for real-time systems. The workshop features a highly interactive format with ample time for in-depth discussions. Topics of interest include:

- different approaches to WCET computation,
- flow analysis for WCET, loop bounds, feasible paths,
- low-level timing analysis, modeling and analysis of processor features,
- strategies to reduce the complexity of WCET analysis,
- integration of WCET and schedulability analysis,
- evaluation, case studies, benchmarks,
- measurement-based WCET analysis,
- tools for WCET analysis,
- program and processor design for timing predictability,
- integration of WCET analysis in development processes,
- compiler optimizations for worst-case paths, and
- WCET analysis for multi-threaded and multi-core systems.

The papers were presented at the workshop were selected based on peer reviews by program committee members and external reviewers. 13 submissions out of 23 were finally selected for presentation. These proceedings contain the presented papers, and the abstract of the invited talk by Dr. Jean Souyris. For the first time a printed version of the final proceedings was printed in advance, and distributed at the workshop, rather than being edited as post-proceedings after the workshop. This version of the proceedings was printed and published by OCG (ISBN 978-3-85403-268-7). The current online version of the proceedings is a re-publication of the printed version.

I am happy to thank the authors, the Program Committee including the external reviewers, the WCET Workshop Steering Committe, and the ECRTS'10 organizers for assembling the components of a very stimulating workshop. The workshop organizers are also deeply grateful to the ArtistDesign Network of Excellence for financial support.

November 2010 Björn Lisper

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

WCET 2010 Steering Committee

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

WCET 2010 Program Committee

Antoine Colin, Rapita Systems Ltd., UK
Amine Marref, Mälardalen University, Sweden
Christine Rochange, IRIT, University of Toulouse, France
Isabelle Puaut, IRISA Rennes, France
Niklas Holsti, Tidorum Ltd., Finland
Stefan Petters, Polytechnic Institute of Porto, Portugal
Heiko Falk, Technische Universität Dortmund, Germany
Chris Healy, Furman University, USA
Raimund Kirner, University of Hertfordshire, UK
Daniel Grund, Saarland University, Germany
Abhik Roychoudhury, National University of Singapore, Singapore
Daniel Kästner, AbsInt GmbH, Germany

WCET 2010 External Reviewers

Sven Bünte
Benjamin Lesage
Michael Zolda
Jose Marinho
Lei Ju
Stephan Wilhelm
Christoph Cullmann
Sudipta Chattopadhyay
Hugues Cassé
Gernot Gebhard
Paul Emberson
Marc Schlickling

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Timing Anomalies Reloaded

Gernot Gebhard[1]

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

Abstract

Computing tight WCET bounds in the presence of timing anomalies – found in almost any modern hardware architecture – is a major challenge of timing analysis.

In this paper, we renew the discussion about timing anomalies, demonstrating that even simple hardware architectures are prone to timing anomalies. We furthermore complete the list of timing-anomalous cache replacement policies, proving that the most-recently-used replacement policy (MRU) also exhibits a domino effect.

1998 ACM Subject Classification B.2.2

Keywords and phrases Timing Anomalies, Domino Effects, MRU Replacement Policy, LEON2

Digital Object Identifier 10.4230/OASIcs.WCET.2010.1

1 Introduction

The validation of the timing behavior of tasks in a safety-critical embedded software system requires both safe and precise worst case execution time (WCET) bounds. Those bounds need to be safe to ensure that each component of the software system performs its job in time. Furthermore, those bounds are required to be precise to ensure the schedulability of such software systems. Two different approaches have emerged to solve the timing analysis problem: measurement-based timing analysis and static WCET analysis. In the following, we focus on static timing analysis and one of the main challenges this analysis method has to face: timing anomalies.

Intuitively spoken, a timing anomaly is a counterintuitive behavior of a hardware architecture, where a “good” event (e.g., a cache hit) leads to an overall longer execution, whereas the opposing “worse” event, such as a cache miss, leads to a globally shorter execution time. In the presence of such anomalies, the local worst case is not always a safe assumption in static timing analysis. To compute safe timing guarantees, any static timing analysis has to consider all possible executions caused by any non-determinism in the abstract hardware model (e.g., such as unknown cache contents). Due to the loss of predictability, the static analysis of architectures featuring timing anomalies requires much more effort in terms of computational power and memory consumption.

Intuitively, one would assume that timing anomalies are restricted to complex hardware architectures. In fact, the Motorola PowerPC 755 is known to have a timing anomaly due to its complex pipeline [10]. Furthermore, architectures with caches with, e.g., PLRU or random replacement policies feature timing anomalies as well [2].

However, even simple architectures can suffer from timing anomalies, as demonstrated throughout this paper. We demonstrate that the LEON2 processor, developed at Aeroflex Gaisler [1], also features a timing anomaly caused by the processor’s cache line fill mechanism.

In addition to discussing the LEON2, we complete the list of commonly used replacement policies that are prone to timing anomalies by examining the MRU replacement policy.

Section 2 discusses related work. Section 3 formally defines timing anomalies and introduces the existing timing anomaly classifications. Section 4 discusses the MRU replacement policy and

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 1–10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

proves that this replacement strategy exhibits a timing anomaly. Section 5 introduces the LEON2 architecture and demonstrates that this architecture is prone to a timing anomaly – despite its rather simple structure. Finally, Section 6 concludes this paper.

2 Related Work

Lundqvist and Stenström [7] are the first to introduce the term timing anomaly. They find that the worst case instruction execution time behavior does not necessarily contribute to the global worst case execution time. In their paper the authors provide an example of a timing anomaly, where a cache hit leads to the worst case timing. Engblom and Jonsson [5] also discuss timing anomalies. They translate the notion of timing anomaly of Lundqvist and Stenström [7] to their model considering (local) timing of pipeline stages instead of whole instructions.

Both Lundqvist and Engblom claim that timing anomalies cannot occur in processors that only comprise in-order resources (i.e., two instructions can only use a resource in program order). This statement is unfortunately not always true, as we show by means of the LEON2 hardware architecture.

Schneider [10] describes a timing anomaly present in the Motorola PowerPC 755 (MPC755). The possibility to dispatch an instruction on two execution units with different behavior in conjunction with pipeline stalls triggers the described timing anomaly.

Thesing [11] discusses the Motorola ColdFire 5307 that has a rather simple in-order pipeline. He shows that the processor exhibits timing anomalies, caused by the pseudo round-robin cache replacement algorithm.

Berg [2] discusses cache replacement policies and their timing anomalies. He finds that caches using first-in first-out (FIFO), round-robin, or pseudo least-recently-used (PLRU) cache replacement strategies suffer from timing anomalies. These replacement strategies are commonly used in embedded hardware architectures, as they require less update logic compared to the LRU policy, which is free of timing anomalies.

In the context of WCET analysis, Reineke et al. [9] provide the first formal definition of timing anomalies. The paper provides a classification of timing anomalies, which we adopt in this paper.

Eisinger et al. [4] provide a novel methodology to automatically detect timing anomalies. Requiring an accurate hardware model to be available (e.g., in VHDL), the approach computes an instruction sequence that triggers a timing anomaly, if such a sequence exists. Yet, the approach is not fully automatic, because hardware features potentially causing timing anomalies need to be identified manually.

Reineke and Sen [8] discuss a related approach. Other than Eisinger et al., they propose a method that allows a static timing analysis to safely discard analysis states by means of Δ functions. A Δ function computes the maximal difference in timing between two system states on any input instruction sequence. For any pair of system states, a static timing analysis can consult the corresponding Δ function to determine which of the two states can be safely discarded. This works well for small hardware models, as demonstrated in the paper, but it remains unclear whether this approach can be applied to complex embedded architectures in a beneficial way.

Kirner et al. [6] show that splitting up a WCET analysis into separate parallel WCET analyses (corresponding to hardware components operating in parallel) is not generally safe in the presence of timing anomalies. Furthermore, the authors identify special instances of “parallel’ timing anomalies still making a parallel decomposition of the WCET problem feasible. Their findings correspond to the classification of architectures (see Section 3.6). Non-fully timing compositional architectures do not allow for a safe, parallel decomposition of the WCET problem.

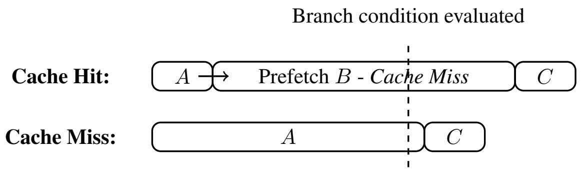


Figure 1 *Speculation-triggered timing anomaly*: The processor executes a conditional branch instruction, whose condition is yet unresolved. Assuming a cache hit for the initial code fetch, the processor speculatively fetches the instruction B that is not contained in the cache. This causes an overall longer execution time, because the cache line fill operation stalls the processor longer than it takes to resolve the branch condition.

3 Timing Anomalies

Intuitively, a timing anomaly is a counterintuitive behavior of a hardware architecture, where a local speed-up leads to a global slow-down. Several – average-performance increasing – hardware features may exhibit this kind of non-local execution time behavior. In the following we formally define timing anomalies in accordance to the definition found in [9]. Furthermore, we discuss some examples of timing anomalies commonly found in modern processors.

3.1 Formal Definition

▶ **Definition 1.** **(Hardware State)** *For a given hardware architecture $\mathcal{A}$, the set $\widehat{\mathcal{H}}$ comprises all possible* hardware states *of that architecture. A specific* hardware state *of the architecture is $\eta \in \widehat{\mathcal{H}}$.*

▶ **Definition 2.** **(Program)** *A* program *P is a directed graph $P = (V, E)$ with $E \subseteq V \times V$, where the nodes V represent instructions, and an edge $(u, v) \in E$ represents the control flow transition from instruction u to v. A* path *$\vec{\pi} = \pi_1\pi_2\ldots$ through the program $P = (V, E)$ is a possibly infinite sequence of instructions, such that $(\pi_i, \pi_{i+1}) \in E$ for each $i < |\vec{\pi}|$.*

▶ **Definition 3.** **(Execution)** *The* execution *$\gamma_{\vec{\pi}}$ of the path $\vec{\pi}$ is a function $\widehat{\mathcal{H}} \times \mathbb{N} \to \mathbb{R}_{\geq 0}$ assigning each instruction on the path $\vec{\pi}$ a non-negative (relative) execution time depending on the initial hardware state η (i.e, $\gamma_{\vec{\pi}}(\eta, i)$ denotes the time the processor needs to execute the instruction π_i). The (absolute)* execution time *under the initial state η until the position $n \in \mathbb{N}$ is $\Gamma_{\vec{\pi}}(\eta, n) = \sum_{i=1}^{n} \gamma_{\vec{\pi}}(\eta, i)$.*

▶ **Definition 4.** **(WCET)** *The* worst case execution time (WCET) *is determined by the worst-case initial hardware state θ, such that $\lim_{n\to|\vec{\pi}|} \Gamma_{\vec{\pi}}(\eta, n) \leq \lim_{n\to|\vec{\pi}|} \Gamma_{\vec{\pi}}(\theta, n)$ for all hardware states η.*

Note that Definition 3 allows for different initial hardware states before the execution of the program. This makes sense, because the precise initial state of the processor is usually unknown in the scope of static timing analysis. Thus static timing analyses need to consider all possible initial hardware states to provide safe WCET bounds.

▶ **Definition 5.** **(Timing Anomaly)** *An architecture has a* timing anomaly *if there exists a path $\vec{\pi}$ through a program P, $i, n \in \mathbb{N}$ with $n > i$, and a hardware state θ, such that $\gamma_{\vec{\pi}}(\theta, i) < \gamma_{\vec{\pi}}(\eta, i)$ and $\Gamma_{\vec{\pi}}(\theta, n) \geq \Gamma_{\vec{\pi}}(\eta, n)$ for all hardware states $\eta \neq \theta$. The state θ is called* timing-anomalous.

3.2 Examples

Figure 1 gives an example of a timing anomaly caused by the interaction between the branch prediction mechanism, the instruction cache, and the processor's ability to execute instructions out-of-order. In

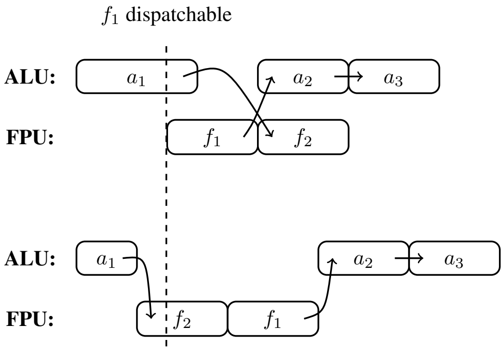


Figure 2 *Variant execution time triggered timing anomaly*: This example demonstrates that a locally fast instruction execution might cause a global slow-down of an instruction sequence. The instructions a_1, a_2, and a_3 execute on the ALU. The ALU features an early-out mechanism that allows integer divide instructions, such as a_1, to complete faster under certain circumstances. The other instructions f_1 and f_2 solely execute on the FPU. Edges between instructions indicate definition-use dependencies.

this example the processor is currently executing a conditional change-of-flow instruction, whose condition is not yet evaluated at the moment the instruction A is about to be fetched. Upon a cache hit for code fetch of instruction A, the processor starts to speculatively fetch the uncached branch target B. Although the initial cache hit locally causes a faster execution, the overall execution is slowed down, because the cache line fill fetching B takes longer than resolving the branch condition.

This timing anomaly could also be caused by speculative execution. This means that the processor starts to execute the fetched instructions, while the processor computes the branch condition. Instead of fetching the instruction B and being stalled due to a cache miss, the processor could speculatively execute the previously fetched instruction B resulting in a longer stall of the processor's pipeline.

Figure 2 demonstrates a different timing anomaly caused by instructions with variant execution times (e.g., due to dividers with an early-out mechanism). Here, the processor features two execution units, an arithmetical logical unit (ALU) and a floating point unit (FPU). Depending on the input parameters, the ALU executes integer division instructions, like a_1, quicker. In this case, completing instruction a_1 earlier, the processor is able to dispatch instruction f_2 in front of instruction f_1. This effectively causes the processor to execute all instructions sequentially. The instruction sequence takes longer to complete, because the processor cannot benefit from its ability to execute instructions in parallel. On the contrary, if instruction a_1 takes longer to complete, the processor will dispatch instruction f_1 earlier. This allows the processor to execute the instructions a_2 and f_2 in parallel, resulting in an overall faster execution.

The variable-execution-time-triggered timing anomaly corresponds to a so-called scheduling anomaly. In the same fashion, a task that terminates earlier could lead to an overall longer schedule. Whereas a faster schedule could be achieved if the very same task would run to completion a bit later. Greedy schedulers are usually unable to prevent this kind of anomaly.

3.3 Domino Effects

The presence of timing anomalies increases the complexity of static timing analyses. A static timing analysis cannot always assume the local worst case, if the analyzed architecture is prone to a timing analysis. Instead the analysis has to take all possibilities into account to compute safe WCET bounds.

Often the effect of a timing anomaly on the execution time stabilizes eventually. This means that any timing-anomalous execution reaches a point where it only differs by a constant from any other execution on the sequence of the input program. Such a timing anomaly is called *k-bounded timing anomaly*, where k is the maximal difference in execution time caused by the timing anomaly. This is formalized in Definition 6. In the presence of a k-bounded timing anomaly, a static timing analysis could always assume the local worst case, adding the constant k to the computed WCET bound [8].[1]

► **Definition 6.** **(k-bounded Timing Anomaly)** *An architecture has a k-*bounded timing anomaly*, if there exists a $k \in \mathbb{R}_{\geq 0}$ such that for all timing-anomalous hardware states θ for the execution of a path $\vec{\pi}$ through a program P holds: $\Gamma_{\vec{\pi}}(\theta, n) - \Gamma_{\vec{\pi}}(\eta, n) \leq k$ for all $n \in \mathbb{N}$ and all states η.*

Unfortunately, some hardware features cause timing anomalies whose effects on timing are unbounded. Such timing anomalies are known as *domino effects*. Domino effects are essentially different from k-bounded timing anomalies: A k-bounded timing anomaly occurring in a loop only has a limited timing effect that eventually stabilizes. In other words, the loop body runtime will only differ for a bounded number of iterations and converge finally. In the presence of a domino effect, the loop body runtime will take different values without convergence in the future.

► **Definition 7.** **(Domino Effect)** *An architecture has a* domino effect*, if it exhibits a timing anomaly that is not k-bounded. Such timing anomalies are also known as* unbounded timing anomalies*.*

Domino effects are real. Schneider [10] has demonstrated that the MPC755 pipeline actually causes a domino effect. Furthermore, Berg [2] was able to show that, in contrast to the LRU replacement policy, the pseudo LRU, the FIFO, and the round-robin replacement strategies suffer from domino effects. Section 4 completes this list, proving the MRU policy to feature a domino effect as well.

3.4 Challenges for Static Timing Analysis

The presence of timing anomalies impacts both performance and precision of a static timing analysis. In general, an analysis must not always choose the locally most expensive execution, as this decision might not always lead to the global worst case execution time. Consequently, the number of states to consider during analysis time increases greatly, if the absence of timing anomalies cannot be proven for an analysis state, where multiple successor states are possible.

Furthermore, the inability of proving the absence of a timing anomaly might also lead to an increase in the computed WCET bound. The timing anomaly discussed in Section 5 can lead to an overestimation of up to 20% (strongly depending on the analyzed program).

3.5 Classification of Timing Anomalies

Reineke et al. [9] discern three different classes of timing anomalies:

- *Scheduling timing anomaly*: Most timing anomalies found in the literature correspond to this class of timing anomalies. Figure 2 is actually an instance of a scheduling timing anomaly. Depending on the execution time of a task, a faster execution might lead to a globally longer schedule. This kind of anomaly is well-known in the scheduling domain and has been thoroughly studied on various scheduling routines.
- *Speculation timing anomaly*: Figure 1 demonstrates such a timing anomaly. An initial cache hit (the local best case) causes a speculative prefetch addressing an instruction that is not cached. The cache miss leads to an overall longer execution. Section 5 discusses a speculation anomaly found in the LEON2 core.

[1] Note that k is an overestimation of the caused effect on timing. In most cases the precision of a static timing analysis will degrade by assuming the local worst case and adding the constant k to the computed WCET bound.

- *Cache timing anomaly*: Cache timing anomalies are caused by some non-LRU cache replacement strategies. Various cache replacement algorithms have been proven to cause domino effects.

The above classification sorts timing anomalies in accordance to the hardware property that is responsible for the timing anomaly. So far, this classification appears to be exhaustive in the sense that it covers all possible hardware features that might trigger timing anomalies.

Yet, the sole knowledge about the timing anomaly class does not suffice for static timing analysis. In addition to the hardware feature causing the anomaly, it is necessary to know the kind of anomaly. A static timing analysis of a hardware architecture that has a k-bounded timing anomaly can be realized with less effort[2] than a static analysis of an architecture that suffers under a domino effect as discussed in Section 3.3.

Currently, it is unclear how to determine whether an instance of a timing anomaly is k-bounded for a certain system state. Depending on the initial hardware state, a hardware feature triggering a domino effect might only cause a constantly-bounded anomaly for this special case. Furthermore, there exists no general approach to compute the constant k for a k-bounded timing anomaly.

3.6 Classification of Architectures

Depending on whether a hardware architecture exhibits k-bounded timing anomalies or domino effects, the architecture can be classified into three categories. Wilhelm et al. introduce the following categorization in [12]:

- *Fully timing compositional architectures:* The architecture does not exhibit any timing anomalies. Hence, the analysis can safely follow local worst-case paths only. The ARM7 is one example architecture of this class. On a timing accident all components of the pipeline are stalled until the accident is resolved. This even allows for a much simpler analysis where architecture components (e.g., cache, bus occupancy, etc.) can be analyzed separately (i.e., a safe parallel decomposition of the WCET problem is feasible).
- *Compositional Architectures with k-bounded effects:* The architecture suffers from k-bounded timing anomalies but not from domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant to the computed WCET bound, as discussed in Section 3.3. So far, no non-fully timing compositional architecture has been formally proven to belong to this class.
- *Non-compositional architectures:* Architectures belonging to this class exhibit domino effects. The MPC755 is known to belong to this class of architectures, because its complex pipeline might cause a domino effect (see Section 3.3). For such architectures timing analyses always have to follow all paths since any local effect may influence the future execution arbitrarily.

4 MRU Domino Effect

This section discusses the most-recently-used replacement strategy in the context of static timing analysis. In contrast to the LRU replacement algorithm, MRU discards the most-recently accessed cache line upon a cache miss. The MRU replacement algorithm is most useful when older data is likely to be reused (e.g., after sequentially scanning an array or a file for data) [3].

Figure 3 demonstrates the domino effect by means of a 2-way cache using the MRU replacement strategy. In this example, the memory locations a and b are accessed in an alternating pattern. Starting with an empty cache, the cache set contents stabilize after two accesses. After the first two accesses

[2] Assuming the constant k is known.

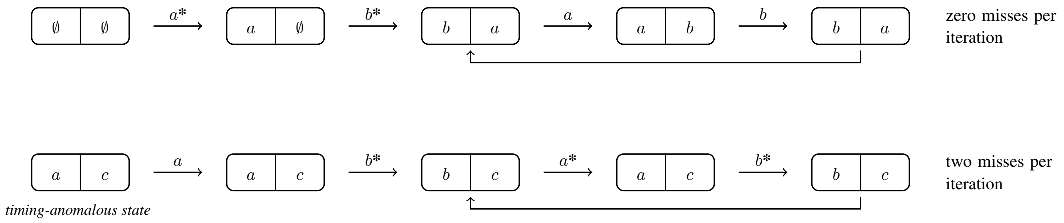


Figure 3 An example domino effect for a 2-way cache using an MRU replacement policy for the repeating access sequence $(a, b)^+$. The left-hand side of a set depicts the most-recently accessed element. The first row features an empty initial cache state, where no misses occur for the given sequence. The second row demonstrates a different initial cache state that causes all accesses except the first to miss the cache. Each miss is marked by *.

that miss the cache set the access sequence will only produce hits. Starting with a cache set that contains the addresses a and c, where a is the most-recently accessed one, each access to the cache except for the first will lead to a cache miss. Because the MRU policy retains older data (i.e., the memory location c in this case), an access to a will evict b from the cache and vice versa.

Proof. (The MRU algorithm exhibits a domino effect) Let $n \in \mathbb{N}_{\geq 2}$ be the associativity of a cache governed under the MRU replacement policy. Additionally, let $h, m \in \mathbb{R}_{\geq 0}$ with $h < m$, where h and m are the costs for cache hit and cache miss, respectively.

To show the presence of a domino effect, we need to find a path through a program and two initial hardware states, such that the difference in execution times starting from the two initial states is not constantly bounded.

Let P be a program alternately accessing the distinct memory locations a and b (starting with a) that map to the same cache set and $\vec{\pi}$ be a path through that program. Furthermore, let η_{empty} be a hardware state where the target cache set is initially empty, and η_{full} be an initial hardware state, where the target cache set contains the disjoint memory locations $a, m_1, m_2, ..., m_{n-1}$ with $m_i \neq b$ for $i \in \{1, 2, ..., n-1\}$ and where the cache line containing a is the most-recently accessed one.

It holds:

$$\gamma_{\vec{\pi}}(\eta_{empty}, i) = \begin{cases} m & i \in \{1, 2\} \\ h & \text{otherwise} \end{cases}$$

$$\gamma_{\vec{\pi}}(\eta_{full}, i) = \begin{cases} h & i = 1 \\ m & \text{otherwise} \end{cases}$$

Furthermore, it holds $\gamma_{\vec{\pi}}(\eta_{full}, 1) < \gamma_{\vec{\pi}}(\eta_{empty}, 1)$ and $\Gamma_{\vec{\pi}}(\eta_{full}, l) \geq \Gamma_{\vec{\pi}}(\eta_{empty}, l)$ for all $l > 2$. Thus, the state η_{full} is timing-anomalous, in accordance to Definition 5.

The timing anomaly is not k-bounded. For any $k \in \mathbb{R}_{\geq 0}$ we can choose an $l \in \mathbb{N}, l > 2$, such that the k-boundedness according to Definition 6 does not hold:

$$\begin{array}{rcll}
 & \Gamma_{\vec{\pi}}(\eta_{full}, l) - \Gamma_{\vec{\pi}}(\eta_{empty}, l) & \leq k & \quad | \quad \Gamma_{\vec{\pi}}(\eta_{full}, 3) - \Gamma_{\vec{\pi}}(\eta_{empty}, 3) = 0 \\
\Leftrightarrow & \sum_{i=4}^{l} (\gamma_{\vec{\pi}}(\eta_{full}, i) - \gamma_{\vec{\pi}}(\eta_{empty}, i)) & \leq k & \\
\Leftrightarrow & \sum_{i=4}^{l} (m - h) & \leq k & \\
\Leftrightarrow & (l - 3)(m - h) & \leq k & \\
\Leftrightarrow & l & \leq \dfrac{k}{m - h} + 3 &
\end{array}$$

For $l > \left\lceil \frac{k}{m-h} \right\rceil + 3$ the above relation is false. ◀

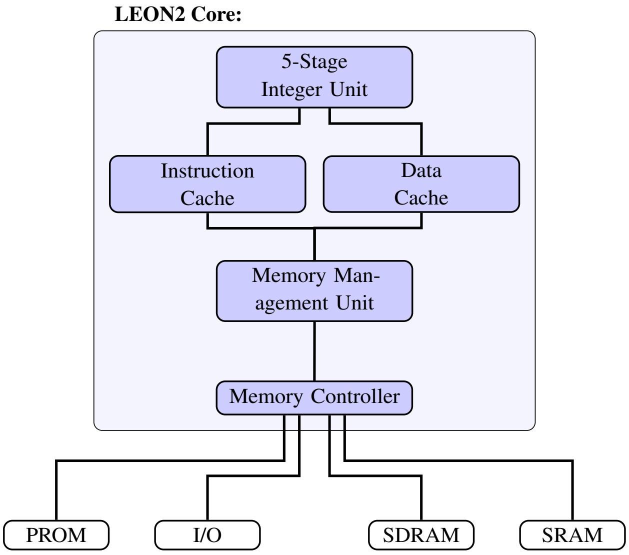


Figure 4 *Simplified block diagram of the LEON2 architecture.*

5 LEON2 Timing Anomaly

In this section we discuss the LEON2 hardware architecture. The LEON2 was developed at Aeroflex Gaisler as a successor of the ERC32 processor. A radiation-hardened version of the LEON2 is available [1] which makes it suitable for the space domain, where fault-tolerance is required.

Similar to the ARM7, the LEON2 features a rather simple pipeline. The pipeline comprises five stages. To speed up execution the LEON2 comprises disjoint instruction and data caches. Figure 4 depicts a block diagram of the LEON2 showing the memory hierarchy.

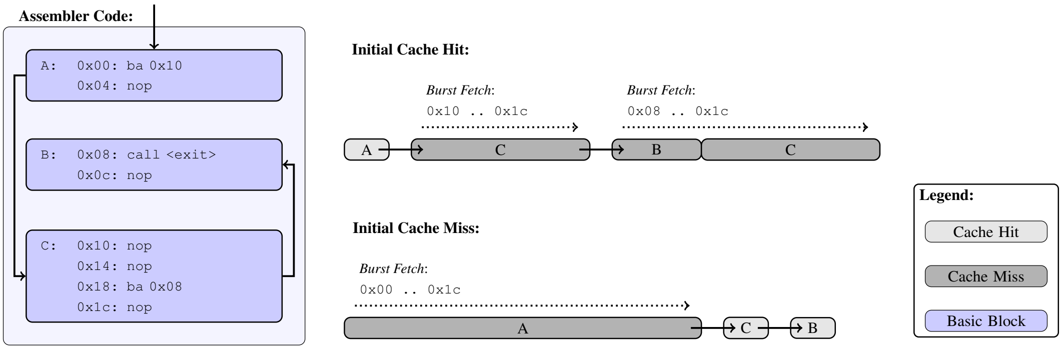
On a first view, the LEON2 appears to be a fully timing compositional architecture. The processor neither performs speculative fetching nor does it execute instructions speculatively. The LEON2 does not possess any branch-prediction mechanism. Instruction are executed and completed in-order. Each instruction has to visit the five pipeline stages one after another. Thus, an instruction cannot overtake a slower instruction blocking a certain pipeline stage. This prevents the possibility of a scheduling anomaly. Upon a timing accident (i.e., a cache miss) the internal pipeline is stalled until the accident is resolved. Both caches commonly use the LRU replacement policy[3], which is known to behave in a timing compositional manner. It appears that none of the above described timing anomalies can occur. However, in the following we will show that the LEON2 has a hardware feature that potentially triggers a timing anomaly (depending on the system state).

Upon a cache miss, the processor needs to load the missing cache line from main memory. Usually, the whole cache line is loaded and put into the cache. Until the cache line has been filled, the processor stalls the originating memory access. To reduce latencies, some architectures start loading the cache line at the requested address directly forwarding the received data to the core (*cache streaming*).

A similar technique is available in the LEON2 architecture. Each cache line is equipped with valid bits for each word[4] inside the cache line. A cache line is either 16 or 32 byte wide and thus

[3] The LEON2 is synthesized from a VHDL model where different replacements algorithms can be configured. Among others, MRU is a possible choice.

[4] A word is four bytes on the LEON2 hardware architecture.



```
A:   0x00: ba 0x10
     0x04: nop

B:   0x08: call <exit>
     0x0c: nop

C:   0x10: nop
     0x14: nop
     0x18: ba 0x08
     0x1c: nop
```

Figure 5 *LEON2 timing anomaly*: The example demonstrates a timing anomaly present in the LEON2 processor caused by the instruction cache line fill mechanism. The basic blocks A, B, and C reside in the same cache line. The local best case — assuming a cache hit for the instructions in basic block A — causes the global worst case execution of the example: The core performs ten instructions fetches. On the contrary, only eight instruction fetches are issued upon an initial cache miss.

comprises either four or eight valid bits. Upon a data cache miss, solely the requested word is loaded from memory and put into the corresponding cache line. The instruction cache operates slightly differently than the data cache. If an instruction fetch misses the code cache, the processor burst-fills the corresponding cache line starting from the requested instruction till the end of the line. The processor does not issue wrap-around burst fetches. Consequently, cache lines might only be filled partially. Furthermore, the processor does not check for existing entries upon burst-filling the cache line. A timing anomaly finally becomes possible, as the LEON2 processor allows cache line fills to be interrupted under certain circumstances.

Figure 5 demonstrates how the described cache line fill mechanism can trigger a timing anomaly. In this example the contents of the cache are assumed to be initially unknown. Each cache line can hold up to eight instructions. Assuming an initial cache miss, the core fills the whole cache line. All in all, the processor issues eight instruction fetches. Assuming cache hits for the first two instruction fetches (basic block A) causes a timing anomaly. The remainder of the target cache line still remains unknown. Reaching the basic block C, a static analysis then would need to assume a cache miss. Recall that the processor might abort a cache line fill operation. Thus, the instructions of basic block C need not necessarily be cached, although cache hits have been assumed for the initial accesses to the cache line. In this case, the core will fill the upper half of the target cache line. Eventually, the program branches to the basic block B. Again, a static analysis would need to assume a cache miss. Because the processor does not check whether burst-fetched instructions are already cached, the instructions in basic block C will be fetched again. Altogether the core performs ten fetches after the initial cache hits. So, the processor performs 20% more memory accesses under the initial assumption.

Despite the simple structure of the LEON2 a timing anomaly is possible, caused by a rather simple, average-case performance increasing hardware feature. Obviously, the timing anomaly is a speculation timing anomaly (see Section 3.5). Fetching subsequent instructions upon an instruction cache miss, the processor assumes a sequential execution of the program.

The described timing anomaly is k-bounded. It is easy to see that the described effect will eventually stabilize – a positive side effect of the employed cache replacement policy. Due to space limitations we can not include the proof in this paper.

The code structure that causes the timing anomaly depicted in Figure 5 is not uncommon. Analyzing industry LEON2 software, we were able to verify that due to the code structure this phenomenon might occur in real world applications. Due to a non-disclosure agreement we must not provide further details about this particular software.

6 Conclusion

In this paper we have proved that the MRU replacement policy is prone to domino effects. By this, we have completed the list of commonly used replacement policies suffering under timing anomalies. Additionally, we have shown that the LEON2 exhibits a timing anomaly, despite its simple structure.

In the future, we plan to study other hardware architectures that are being used in the automotive and the aerospace domain. Furthermore, we will check whether known instances of timing anomalies can be proven to be k-bounded.

References

1 AEROFLEX GAISLER. `http://www.gaisler.com`.

2 Christoph Berg. PLRU cache domino effects. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden*, number 06902 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), July 2006.

3 Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB 1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 127–141. VLDB Endowment, 1985.

4 Jochen Eisinger, Ilia Polian, Björn Becker, Stephan Thesing, Reinhard Wilhelm, and Alexander Metzner. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *DDECS '06: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, pages 13–18, Washington, DC, USA, 2006. IEEE Computer Society.

5 Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static WCET analysis. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 334–348, London, UK, 2002. Springer-Verlag.

6 Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 119–128, Dublin, Ireland, July 2009. IEEE.

7 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.

8 Jan Reineke and Rathijit Sen. Sound and efficient wcet analysis in the presence of timing anomalies. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

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

10 Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

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

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

Bounding the Effects of Resource Access Protocols on Cache Behavior

Enrico Mezzetti[1], Marco Panunzio[1], and Tullio Vardanega[1]

**1 University of Padua, Department of Pure and Applied Mathematics,
via Trieste, 63 35121 Padua, Italy**
{emezzett,panunzio,tullio.vardanega}@math.unipd.it

Abstract

The assumption of task independence has long been consubstantial with the formulation of many schedulability analysis techniques. That assumption is evidently advantageous for the mathematical formulation of the analysis equations, but ill fit to capture the actual behavior of the system. Resource sharing is one of the system design dimensions that break the assumption of task independence. By shaking the very foundations of the real-time analysis theory, the advent of multicore systems has caused resurgence of interest in resource sharing and synchronization protocols, and also dawned the fact that the assumption of task independence may be forever broken. Research in cache-aware schedulability analysis instead has paid very little attention to the impact that synchronization protocols may have on cache behavior. A blocked task may in fact incur time penalties similar in kind to those caused by preemption, in that some useful code or data already loaded in the cache may be evicted while the task is blocked. In this paper we characterize the sources of cache-related blocking delay (CRBD). We then provide a bound on the CRBD for three synchronization protocols of interest. The comparison between these bounds provides striking evidence that an informed choice of the synchronization protocol helps contain the perturbing effects of blocking on the cache state.

Keywords and phrases Resource access protocols, cache, worst-case response time

Digital Object Identifier 10.4230/OASIcs.WCET.2010.11

1 Introduction

The correctness of schedulability analysis techniques for preemptive real-time systems relies on the use of safe estimates of both the worst-case execution time (WCET) of the system tasks and the additional costs due to interrupts and task preemptions. Whereas the determination of safe and tight WCET bounds is a widely acknowledged and studied problem [14], most schedulability analysis techniques rest on the simplifying assumption of constant (and negligible) context-switch costs. Unfortunately, the use of hardware acceleration features like caches and complex pipelines breaks this assumption for good. With the adoption of caches, in particular, the context-switch cost is no longer constant as it must account for the interferences between tasks: interrupt handling and preemption may influence the execution time of a preempted task. On resumption in fact, the preempted task may incur a number of additional cache misses as some useful cache contents may have been evicted from the cache. Cache-aware schedulability analysis techniques [5, 12, 13] aim at including those cache effects in the schedulability analysis by accounting for the so-called cache-related preemption delay (CRPD) overheads in the response time of individual tasks.

However, interference caused by tasks is not limited to task preemptions. The assumption of task independence rarely holds in practice and real systems often include shared resources which multiple tasks can access in mutual exclusion. Task blocking therefore occurs, which

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 11–22
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

causes priority inversion to arise, and the need for resource access protocols to bound it. In response time analysis (RTA) for fixed-priority systems, the time a task is forced to wait for a shared resource in use by lower priority tasks (the task *blocking time*) is assumed to be bounded.

When it comes to cache interference, however, task blocking may cause effects similar in kind to task preemption, in that some useful code or data blocks already loaded in the cache may be evicted while the task is being blocked. Very few works [10] consider the additional time spent in reloading the evicted cache contents, which is referred to as Cache-Related Blocking Delay (CRBD). Although a task typically suffers from blocking as it shares some resources with lower priority tasks, different patterns and durations of blocking – and thus the amplitude of the CRBD – can be induced by the specific resource access policy in use.

We contend that the CRBD cannot be dismissed as a negligible cache-related effect, and should instead be accounted for by cache-aware schedulability analysis. The contribution of this paper is a characterization of the effects of blocking on the cache behavior (i.e., CRBD) in fixed-priority preemptive systems and the formalization of a worst-case bound of the incurred delay under different resource access protocols. The rest of the paper is organized as follows: section 2 surveys the state-of-the art approaches to cope with cache interference between tasks; section 3 provides a formal definition of the CRBD and defines a bound to it for three resource access protocols of interest; section 4 finally draws some conclusions.

2 Related Work

In general, WCET analysis approaches focus on intra-task cache behavior and, while not directly accounting for inter-task (i.e., extrinsic) interference, try to include the cache effects of the latter in schedulability analysis. In this paper we do not consider limited-preemptible systems, cache partitioning or locking techniques, which may attenuate inter-task interference.

Since task preemption is typically regarded as the main source of interference between tasks, the inclusion of cache effects in schedulability analysis is generally accomplished by accounting for an upper bound on the CRPD in the response time of individual tasks. For example, the RTA iterative equation for task τ_i has been extended to include cache effects due to preemptions as follows:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{1}$$

where w_i^k refers to the time window under analysis, C_i and B_i are respectively the WCET and maximum blocking time for the analyzed task, and the remaining term represents the *interference* from higher priority tasks, which includes the induced CRPD γ_j [5].

The exact CRPD depends on both the preempted and the preempting tasks since it captures the time required by the preempted task to reload cache blocks that have been evicted by the preempting tasks and that will be reused when the preempted task resumes execution.

Two basic concepts are useful to understand the bounds on cache interference between tasks:

- *Useful Cache Blocks* (UCB): cache blocks that may be referenced again before they could be evicted by other memory blocks, according to the cache replacement policy;
- *Used Cache Blocks* ($\overline{UCB}$): cache blocks that may be accessed during the execution of the preempting task.

Earlier approaches [5, 6] that relied on UCB or $\overline{UCB}$ alone to compute the CRPD bound were overly conservative as neither all UCB would be always evicted nor the $\overline{UCB}$ would necessarily evict useful blocks. Less pessimistic approaches have been suggested in [7, 13, 12], which account for both UCB and $\overline{UCB}$ to compute the CRPD bound. Moreover, acknowledging the fact that a cache block may not be useful (or used) along every program path, Negi et al. [9] introduce the more elaborate notions of Cache Utility Vector (CUV) and Final Usage Vector (FUV) to capture all possible cache states along the different execution paths of both the preempted and the preempting tasks. A refinement in UCB computation, combined with WCET analysis, has been proposed in [1]. More recently, with respect to LRU set-associative caches, the idea of *resilience* has been introduced [2] to exclude from the CRPD computation those UCB that can be guaranteed to persist in the cache, thanks to the specific replacement policy.

The term B_i in Equation 1 refers to an upper bound to the blocking time suffered by task τ_i due to resource contention. As observed in [10], similarly to the interference from higher priority tasks, the interference from lower priority tasks contending for shared resources should be considered to predict cache behavior. The effects of blocking are similar to those related to the CRPD since a lower priority task may evict some useful cache blocks of a higher priority task, which thus incurs some CRBD.

The work in [10] extends previous work by the same authors on fully-preemptive and non-preemptive task regions, to cope with shared resources under the Priority Inheritance Protocol. They use a complex framework that exploits task phasing to account for both CRPD and CRBD in the response time of a task. The computation of the CRBD (limited to data cache) employs the same concepts as used to compute the CRPD. In contrast to [10], in this paper we focus on computing an upper bound on the CRBD under different protocols, rather than on its inclusion in the worst-case response time analysis.

3 Cache-Related Blocking Delay

Shared resources typically need to be accessed in mutual exclusion. When a high priority task needs to access a resource that is already locked by a lower priority task, it cannot proceed until the lower priority task completes execution inside the resource and relinquishes its lock. Whenever a lower priority task prevents the execution of a higher priority task, the system experiences potentially unbounded priority inversion. This phenomenon can be bounded with the use of a resource access protocol. In this paper we focus on three well-known protocols: the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP) and the Immediate Ceiling Protocol (ICP).

In a fixed-priority preemptive system with shared resources equipped with a synchronization protocol, three different kinds of blocking may arise [8]:

- *Direct blocking* occurs when a higher priority task requests a shared resource held by a lower priority task; another form of direct blocking, *transitive blocking*[1], occurs when nested resources access is permitted, and a blocked task transitively suffers from the blocking incurred by the blocking task itself.
- *Inheritance or push-through blocking* occurs for a task τ_m that does not need any shared resource, when a lower priority task τ_j blocks a task τ_i with priority $\pi(\tau_i) > \pi(\tau_m)$ and executes at a priority higher than $\pi(\tau_m)$ due to some priority inheritance rule.

[1] Also referred to as *chain blocking*.

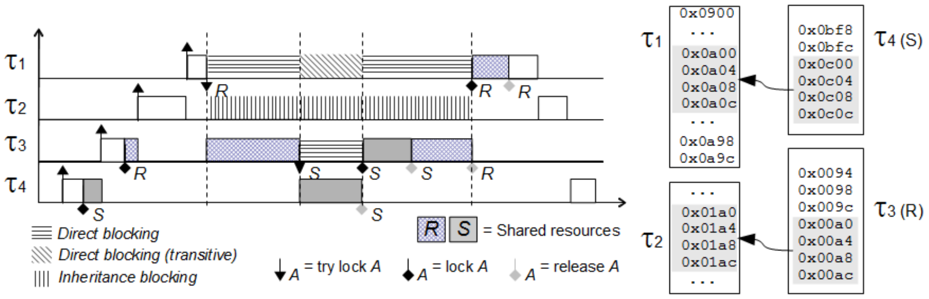


Figure 1 Example of CRBD.

- *Avoidance blocking* occurs when a task τ_i is denied access to an *available* resource to prevent deadlock.

From the standpoint of caches, the execution of the blocking task inside a critical section may cause the eviction of useful cache contents that would have later been reused by the blocked task. A high priority task will thus incur a time penalty (or blocking delay) because of the additional cache misses, regardless of the type of suffered blocking.

The scenario depicted in Figure 1 illustrates the different types of blocking under the PIP and shows how lower priority tasks may affect the cache content of higher priority tasks. In particular, task τ_1 and τ_3 suffer direct blocking, while task τ_2 suffers inheritance blocking. Assume that τ_1 has loaded four cache blocks that would be shortly reused (i.e., the shaded memory addresses in Figure 1) in a small direct mapped instruction cache. Unfortunately, task τ_1 is blocked when trying to access shared resource R currently held by τ_3, which in turns is blocked by τ_4 on the shared resource S. While τ_3 has no effect on the useful cache content of τ_1, the code executed by τ_4 in its critical section accessing S maps exactly to the same cache sets and evicts all the four useful blocks of τ_1. When τ_1 resumes, it will incur a CRBD of four additional cache misses.

A subtler penalty is experienced by task τ_2 due to the execution of τ_3: whereas it does not share any resource with other tasks, τ_2 is blocked due to priority inheritance. In the example, the useful cache content of τ_2 is evicted during the execution of τ_3 inside its critical region. It is worth noting that τ_2 would not have suffered any interference (CRPD) due to the higher priority task τ_1.

Hence, blocking does not only affect the response time of a task as a single worst-case factor independent of the task itself, but it may also directly affect its execution time. This is so because priority inversion, even if bounded, causes a cache-related delay akin to that caused by preemption. Similarly to CRPD, the amount of delay potentially incurred by a task on a single blocking event depends on both the cache content of the blocked task and the execution of the blocking task.

The actual CRBD is thus a function of the UCB (cf. Section 2) of the blocked tasks and the $\overline{UCB}$ of the blocking task. In contrast to CRPD, however, the $\overline{UCB}$ are not determined by the whole execution of the blocking task since the induced delay can stem just from the execution inside critical sections. Furthermore, in case of direct (and some forms of avoidance) blocking, also the UCB of the blocked task are limited to those determined at the beginning of the critical section at which the task is blocked.

Although we can expect the CRBD to be small for single critical sections, its relevance increases as soon as a task may experience several and potentially different blocking events during the same activation. The CRBD arising from such blocking events cannot be disregarded during schedulability analysis as its cumulative effect may invalidate the analysis results.

We performed some initial experiments to gage by static analysis the impact of the CRBD on the instruction cache performance of blocked tasks. We implemented a small test case made up of three tasks: τ_1 and τ_3, reading and/or writing a shared resource, and a notionally independent task τ_2, taken from the Mälardalen benchmark[2]. We extended the Heptane tool from IRISA/Rennes to compute the number of additional cache misses incurred by τ_1 and τ_2 owing to direct blocking and indirect blocking respectively. With a 4 KB, 32 B lines, direct-mapped instruction cache, under PIP, task τ_1, whose stand-alone cache performance shows 30 misses over 1011 cache accesses, may suffer 8 additional cache misses from the CRPD directly induced by τ_3. Task τ_2 may incur 3 further cache misses against a stand-alone cache performance of 9 misses over 54 accesses, when the impact of inheritance blocking is not considered[3].

The CRBD potentially suffered from a task depends on the actual resource access protocol in use, as it determines both the possible types of blocking incurred and the maximum number of blocking events suffered for each activation.

In principle, the CRBD problem could be transformed into a CRPD problem by modeling the critical sections as tasks that may preempt higher priority tasks, thus exploiting the intrinsic similarities between the two phenomena. However, some specialization should still be needed to capture the specificity of blocking as well as of the resource access protocol in force, thus boosting the complexity of the analysis approach considerably. For example, preemption points should be predefined to permit a sound computation of UCB in case of direct blocking. Similarly, one may need to account for the transitivity of blocking depending on the resource access protocols in use. In our approach, instead, we solely focus on the computation of UCB and $\overline{UCB}$ and exploit sound theoretical bounds [11] on the number of blocking events to provide an upper bound on the CRBD.

In the following we first provide a formal characterization of the CRBD potentially incurred by a task; and then we exploit well-known bounds on the number of blocking events suffered by a task, under different resource access protocols to observe that the way in which the worst-case CRBD can affect the execution time of tasks is highly related to the choice of protocol.

3.1 Bounding the Cache-Related Blocking Delay

We assume total ordering between tasks such that $i < j$ if $\pi(\tau_i) > \pi(\tau_j)$: hence τ_0 is the highest priority task. In our model a task τ_i self-suspends only at the end of every execution of its jobs, and may access a shared resource $R \in SR_i$, where SR_i identifies the subset of the system resources (SR) that get accessed by τ_i.

It is worth noting that from a finer-grained point of view, acquiring (respectively releasing) a shared resource corresponds to entering (exiting) a critical section where the resource is locked (unlocked). Since a task τ_i may access a shared resource R through different critical sections, we define cs_i^R to be the set of critical sections in τ_i accessing the resource $R \in SR_i$. Similarly, $cs_{i,k}^R$ identifies the k^{th} critical section in τ_i accessing the resource $R \in SR_i$. In any case, we assume critical sections to be properly nested so that they can never overlap. For every pair of critical sections $cs_{i,k}$, $cs_{i,z}$ in τ_i either $cs_{i,k} \subset cs_{i,z}$,

[2] *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*
[3] Furthermore, in this case, no additional misses originate from preemption by τ_1.

$cs_{i,k} \supset cs_{i,z}$ or $cs_{i,k} \cap cs_{i,z} = \emptyset$.

The determination of the CRBD incurred by a task exploits similar concepts as when computing the CRPD, involving the computation of UCB and $\overline{UCB}$ for blocked and blocking task respectively. First, we recall that the set of UCB and $\overline{UCB}$ for a task τ_i are dependent on each specific node n in the Control-Flow Graph (CFG) of τ_i, where each node represents a basic block. In fact, UCB and $\overline{UCB}$ can be safely computed at basic block level, as proved in [6].

According to [7, 13, 12] the UCB_i^n for a task τ_i at node n can be computed as the intersection between the sets of *ReachingBlocks* (RB) and *LiveBlocks* (LB) at node n where RB is the set of cache blocks potentially cached at node N, whereas LB is the set of blocks that could potentially be reused in the successors of n. Intuitively, instead, $\overline{UCB}_i^n$ can be computed as $RB_i(n)$. Thus, $UCB_i^n = RB_i(n) \bigcap LB_i(n)$ and $\overline{UCB}_i^n = RB_i(n)$.

In case of blocking, we are interested in determining UCB and $\overline{UCB}$ for a task τ_i blocked on a critical section $cs_{i,k}^R$. For example, let us consider a simple case of direct blocking between two tasks. Task τ_i is blocked when trying to access critical section $cs_{i,k}^R$ because a lower-priority task τ_j is executing inside a critical section $cs \in cs_j^R$ accessing the same shared resource R. In this case, the set of UCB for the blocked task τ_i is to be computed with respect to the node n_R trying to enter $cs_{i,k}^R$.

$$UCB_{i,k}^R = RB_i(n_R) \cap LB_i(n_R), \text{ where } n_R \text{ is the entry node of } cs_{i,k}^R$$

The set of $\overline{UCB}$ for the blocking task τ_j must be computed with respect to the critical section $cs_{j,h}^R$ it is executing within, as only the RBs in $cs_{j,h}^R$ can affect the cache state of τ_i. For this reason, we extend the notion of RB to address *intervals* of nodes in the CFG instead of single nodes.

Given an interval $[n_1, n_2] = \mathcal{I} \in CFG(\tau_i)$, we define $RB_i(\mathcal{I})$ as the contribution to $RB(n_2)$ of all possible paths in $CGF(\tau_i)$ from node n_1 to n_2. Accordingly,

$$\overline{UCB}_j(cs_{j,h}^R) = RB_j(cs_{j,h}^R) = RB_j([first\_node, last\_node]_{cs_{j,h}^R})$$

In the example, the execution of τ_j inside $cs_{j,h}^R$ may evict some useful cache blocks that τ_i may have loaded in the cache before its attempt to enter $cs_{i,k}^R$. The incurred CRBD can be computed as a function of the $UCB_{i,k}^R$ and $\overline{UCB}_{j,h}^R$ terms just defined:

$$CRBD = \otimes_\sigma\big(UCB_{i,k}^R, \overline{UCB}_j(cs_{j,h}^R)\big) \times \text{miss penalty} \tag{2}$$

where the $\otimes_\sigma$ operator accounts for the actual cache associativity and replacement policy in combining the information on useful and used cache blocks, cf. [13, 1]. For example, for direct-mapped caches, $\otimes_{DM}(UCB, \overline{UCB})$ will include those cache sets which at least one cache block in both UCB and $\overline{UCB}$ is mapped to (set intersection). For LRU n-way set-associative caches, instead, the $\otimes_{LRU}$ operator must account for the number of additional cache misses for each cache set. In case of a non-empty $\overline{UCB}$ set, those misses are bounded by the minimum between the cache associativity (n) and the number of UCB mapping to that cache set [4].

In case τ_i and τ_j share more than one resource, we can generalize Equation 2 to determine an upper bound on the delay suffered by τ_i, due to a *single* direct blocking by τ_j for any critical section accessing any shared resource as follows:

$$CRBD_{i,j} \leq \max_{\substack{R \in SR_i, k \in [1, |cs_i^R|] \\ cs \in cs_j^R}} \big\{ \otimes_\sigma \big(UCB_{i,k}^R, \overline{UCB}_j(cs)\big)\big\} \times \text{miss penalty} \tag{3}$$

However, Equation 3 just holds in this simple case where neither transitive direct blocking nor other types of blocking are taken into account. In terms of CRBD, determining the effects of inheritance blocking is much more complex, as the computation of UCB for the blocked task cannot make any simplifying assumption on when the task actually gets blocked.

A more comprehensive bound on the CRBD incurred by a task can be computed by leveraging on the bounds that a specific resource access protocol places on blocking. An upper bound on the worst-case number of blocking events incurred by a task is given in [11, 3] for each protocol. That bound is then combined with the worst-case duration of each critical section to derive a bound on the blocking time potentially suffered by a task. Those bounds typically rely on the notion of potentially blocking critical sections to account for any type of blocking that may occur under the protocol itself. To this end, $\beta_{i,j}$ is defined in [11] as the set of critical sections of a lower-priority task τ_j which can block τ_i in any way. The bounds on the number of blocking events and blocking time exploit the $\beta^*_{i,j}$ set which identifies the set of *outermost* critical sections of τ_j that can block τ_i. More formally:

$$\beta^*_{i,j} = \{(cs_{j,k} | cs_{j,k} \in \beta_{i,j}) \wedge (\neg \exists cs_{j,m} \in \beta_{i,j}, cs_{j,k} \subset cs_{j,m})\}$$

We will exploit the same concepts, with the only difference that we are not interested in the critical section that may incur the maximum blocking time since we focus on the CRBD, which is independent of the duration of the critical section. Instead, we are interested in the critical section $cs_{j,k} \in \beta^*_{i,j}$ which causes the eviction of the greatest number of useful blocks for the blocked task, for all lower-priority tasks τ_j.

In the following, we will combine given bounds on the number of blocking events with the same concepts as used in CRPD analysis to provide a *safe* upper bound on the CRBD under different protocols.

3.2 CRBD under the Priority Inheritance Protocol

When access to shared resources is managed with PIP [11], whenever a task that holds the lock of a resource blocks a higher-priority task, it inherits the priority of the highest-priority task it is blocking[4]. When a task releases the lock of a resource, its priority is lowered to the *highest inherited* priority value[5] [15].

PIP is interesting as it does bound priority inversion and also does not require any knowledge on the system's tasks and their priorities, since the priority value to inherit is determined dynamically. Unfortunately, PIP does not prevent deadlock (which may occur in case of nested critical sections) and a task can be blocked multiple times during a single activation. In fact, a task τ_i can be blocked for the duration of at most $\min(n, m)$ outermost critical sections, where n is the number of lower-priority tasks that may block τ_i and m is the number of semaphores[6] that can be used to block τ_i. In the following we re-elaborate both bounds from the standpoint of the CRBD.

3.2.1 Bound on Lower Priority Tasks

Under PIP, a high priority task τ_H can be blocked by a lower priority task τ_L for at most the duration of one critical section of $\beta^*_{H,L}$. Therefore, given a task τ_i for which there are

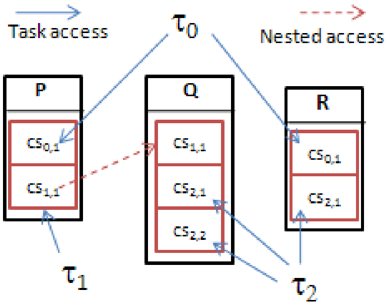
[4] This occurs to transitively inherit priority in case of chain blocking.

[5] The original protocol restores the priority to the value inherited before entering the critical section, which is incorrect.

[6] A semaphore corresponds to a shared resource since we assume each resource to be guarded by a binary semaphore.

n lower priority tasks $\{\tau_{i+1}, \ldots, \tau_{i+n}\}$, τ_i can be blocked for at most the duration of one critical section in each $\beta^*_{i,k}$, $i+1 \le k \le i+n$ [11].

If we assume that all shared resources and critical sections are statically known, we can define a resource access graph and table, similar to that shown in Figure 2.



	P	Q	R
τ_0	$cs^P_{0,1}$		$cs^R_{0,1}$
τ_1	$cs^P_{1,1}$	$cs^Q_{1,1}$	
τ_2		$cs^Q_{2,1}$ $cs^Q_{2,2}$	$cs^R_{2,1}$

Figure 2 Resource graph and corresponding resource access table.

Note that the critical section $cs_{1,1}$ of resource P performs a nested access to critical section $cs_{1,1}$ of resource Q. In this case, the $\beta_{i,j}$ sets derived from Table 2 are as follows: $\beta_{0,1} = \{cs^P_{1,1}, cs^Q_{1,1}\}$ due to resource nesting, $\beta_{1,2} = \{cs^Q_{2,1}, cs^Q_{2,2}, cs^R_{2,1}\}$ (by inheritance blocking), and $\beta_{0,2} = \{cs^R_{2,1}, cs^Q_{2,1}, cs^Q_{2,2}\}$ as τ_2 could transitively block τ_0 by blocking τ_1. The $\beta^*_{i,j}$ sets, instead, removes redundant innermost critical sections; thus, for example, $\beta^*_{0,1} = \{cs^P_{1,1}\}$.

As discussed in Section 3.1, computing the UCB of the blocked task τ_i in case of inheritance blocking needs to consider any possible node in $CFG(\tau_i)$, similarly to task preemption. To avoid the overestimation in considering all possible nodes, we will threat inheritance blocking separately.

An upper bound on the CRBD in case of direct blocking of τ_i due to τ_j is the maximum $\otimes_\sigma$ applied to UCB and $\overline{UCB}$ for any resource accessed by τ_i, every critical section in τ_i accessing that resource and every outermost critical section of τ_j potentially blocking τ_i. Hence, it can be formalized as:

$$CRBD^{base}_{i,j} \le \max_{\substack{R \in SR_i, k \in [1, |cs^R_i|] \\ cs \in \beta^*_{i,j}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{4}$$

With regard to inheritance blocking, we need to account for the most penalizing blocking point for τ_i (i.e., node in the CFG). To this end we define $\widehat{\beta}_{i,j}$, a subset of $\beta^*_{i,j}$ including all critical sections in τ_j which can block τ_i due to inheritance blocking. Thus, $\widehat{\beta}_{i,j} = \{cs | cs \in \beta^*_{i,j} \wedge cs$ can block τ_i due to inheritance blocking$\}$. We can now compute the maximum CRBD incurred by τ_i due to inheritance blocking by τ_j as follows:

$$CRBD^{inherit}_{i,j} \le \max_{\substack{cs \in \widehat{\beta}_{i,j} \\ n \in CFG(\tau_i)}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{5}$$

However, since a lower priority task τ_j can block τ_i because it is executing inside at most one $cs \in \beta^*_{i,j}$, each τ_j can induce solely one of either inheritance or "non-inheritance" blocking on τ_i. Hence, we can safely account for the worst-case blocking (inheritance or not), that is:

$$CRBD_i \le \sum_{j > i} \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \tag{6}$$

3.2.2 Bound on Semaphores

A second upper bound on blocking, based on the number of semaphores potentially blocking a task under PIP is given in [11]. Under PIP, if there are m semaphores which can block task τ_i, then τ_i can be blocked at most m times, as it can be blocked at most by one critical section for each potentially blocking semaphore. Since we assume that each semaphore corresponds exactly to a shared resource, then τ_i can be blocked at most by one critical section for each potentially blocking resource.

Similarly to the previous case, [11] defines $\xi_{i,j,k}$ as the set of critical sections of a lower-priority task τ_j guarded by a semaphore S_k and which can block τ_i (due to any type of blocking). Subsequently, $\xi^*_{i,j,k}$ identifies the set of all potentially blocking outermost critical sections guarded by S_k, that is $\xi^*_{i,j,k} = \{cs^{S_k}_{j,m} | cs^{S_k}_{j,m} \in \beta^*_{i,j}\}$.

For example, recalling Table 2, $\xi^*_{0,1,P} = \{cs^P_{1,1}\}$, $\xi^*_{0,1,Q} = \{cs^Q_{1,1}\}$, $\xi^*_{0,2,R} = \{cs^R_{2,1}\}$ and $\xi^*_{1,\cdot,R} = \{cs^R_{2,1}\}$ (inheritance). Similarly to the first bound, we define $\widehat{\xi}_{i,j,k}$, a subset of $\xi^*_{i,j,k}$ including all critical sections in $\xi^*_{i,j,k}$ guarded by the semaphore S_k which can block τ_i through inheritance blocking. Thus, $\widehat{\xi}_{i,j,k} = \{cs | cs \in \xi^*_{i,j,k} \wedge cs$ can block τ_i due to inheritance blocking$\}$ can be used to separately account for the direct and inheritance cases. First we provide a means to compute the maximum CRBD for each resource that accounts for any lower priority task and any cs in those tasks that may incur both forms of blocking.

$$CRBD^{base}_{i,R} \leq \max_{\substack{j>i, k\in[1,|cs^R_i|] \\ cs\in\xi^*_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{7}$$

$$CRBD^{inherit}_{i,R} \leq \max_{\substack{n\in CGF(\tau_i) \\ j>i \\ cs\in\widehat{\xi}_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{8}$$

Again, since task τ_i can be blocked at most once for each semaphore (resource), we can compute a safe upper bound on the blocking delay by summing the $|S|$ worst-case penalties over the $S \subset SR$ semaphores (resources) potentially blocking τ_i:

$$CRBD_i \leq \sum_{R\in S} \max \left(CRBD^{base}_{i,R}, CRBD^{inherit}_{i,R} \right) \tag{9}$$

The actual bound on the CRBD under PIP is then determined by the minimum between the bounds on lower priority tasks and semaphores (i.e., Equations 6 and 9).

3.3 CRBD under the Priority Ceiling Protocol

With PCP [11], each resource is assigned a *ceiling priority* which is set to at least the priority value of the highest-priority task that uses that resource. Since ceiling priorities are assigned statically, all the tasks of the system and their priority must be known statically. For a task τ_i to be able to access the critical section of a resource, its current priority must be higher than the ceiling priority of any currently locked resource (i.e. semaphore). Otherwise, the task that blocks τ_i inherits the ceiling priority of the resource it is locking.

PCP introduces *avoidance blocking*: a task, when trying to access a resource that is *currently available*, is blocked if its current priority is not higher than the highest ceiling of all semaphores currently locked by other tasks. This protocol rule is used to warrant the absence of deadlock. Furthermore, transitive blocking is not possible, a task τ_i can be blocked at most once per activation, and the duration of the priority inversion is minimized.

Similarly to PIP, a bound on the delay incurred by the effects of blocking on the cache state must account for inheritance blocking separately from direct and avoidance blocking as only the latter ones are triggered when a task attempts to access a resource. Provided that the computation of the $\beta^*_{i,j}$ set includes all critical sections of τ_j that may block[7] τ_i due to direct, inheritance or avoidance blocking, an upper bound for the CRPD can be computed in a similar way to the first bound on PIP. The CRBD suffered by a task τ_i can be bounded by the following equation:

$$CRBD_i \leq \max_{j>i} \left\{ \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \right\} \tag{10}$$

where $CRBD^{base}_{i,j}$ and $CRBD^{inherit}_{i,j}$ are exactly as defined in the PIP case (Eq. 4 and 5 respectively). As opposed to the PIP case, we are interested just in the most penalizing critical section among all critical sections and all lower-priority tasks, due to Theorem 12 in [11].

3.4 CRBD under the Immediate Ceiling Priority Protocol

The Immediate Ceiling Priority Protocol (ICPP) (direct derivative of Baker's stack resource policy [3]) is similar to PCP, as ceiling priorities are assigned to resources with the same rules. Under ICPP however, a task that enters in a critical section always inherits the ceiling priority, while under PCP only when it is *blocking* a higher-priority task; therefore all tasks with a priority lower than or equal to the ceiling priority cannot be scheduled until the resource has been released. ICPP retains the advantages of PCP: absence of deadlock, tasks can block at most once during each activation and the blocking duration is minimized.

The maximum blocking time for a task τ_i is bounded by the longest outermost critical section executed by a lower-priority task τ_j using a resource with a ceiling priority greater than or equal to the priority of τ_i.

More importantly from the CRBD standpoint, the rules of ICPP prevent any disturbing effects on the cache state of the blocked task. In fact, if blocking occurs, it is always before the affected job begins execution; this implies that cache analysis does not need to account for any effect and can continue to assume the worst-case initial cache state (empty or chaos state, depending on the analysis approach). More formally: $CRBD_i = 0$, $\forall \tau_i$.

3.5 Including the CRBD in Response Time Analysis

A safe bound β_i on the CRBD suffered from each task can be straightforwardly included in the iterative equation of response time analysis. In contrast to CRPD, the delay incurred by blocking does not need to propagate to lower priority tasks since it is separately considered for each task:

$$w_i^{n+1} = C_i + B_i + \beta_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{11}$$

where both B_i and β_i depend on the resource access protocol of choice. It is worth noting that the worst-case blocking time and CRBD are not guaranteed to occur altogether. In principle, it could be possible to tighten the computation by accounting for the maximum co-occurrence of B_i and β_i.

[7] Note that the ceiling priority of the resource must be considered when determining potentially blocking critical sections.

As seed for reflection, we note that when it comes to more complex analysis approaches, like e.g., resilience analysis for set-associative caches [2], computing the β_i and γ_j terms separately may *invalidate* the CRPD analysis result, as blocking may incur additional accesses to a cache set.

The comparison of the bounds obtained for the protocols addressed in this paper, though limitedly to their bounds on lower priority tasks[8], shows that ICP is by far preferable with respect to interference on cache as it does not incur any CRBD. The CRBD bounds we provided are pessimistic. Tighter bounds could be computed by straightforwardly extending our approach to a more precise representation for UCB and $\overline{UCB}$ like in [9] or by taking advantage, for example, of task phasing.

4 Conclusion

In this paper we contended that the cache effects caused by the use of synchronization protocols to arbitrate the access to shared resources cannot be dismissed as negligible. Cache contents that are useful to a task of interest may in fact be evicted by lower-priority tasks when the task is blocked. Moreover, different protocols may incur different effects on the task state of the blocked task.

We provided a (pessimistic) bound on the cache-related blocking delay for two well-known protocols: the Priority Inheritance Protocol and the Priority Ceiling Protocol. We also showed that the use of the Immediate Ceiling Protocol does not induce any CRBD, as tasks can be blocked only once per activation and prior to their execution after release.

Although the quantitative effect of the CRBD is not likely to compare with the CRPD, it should not be dismissed as irrelevant: it is arguably important to include it in schedulability analysis that aims to accuracy. We also contend that the cache-related impact should also be contemplated as a distinct evaluation criterion for the selection of the resource access protocol to adopt in a real-time system.

In future work, we plan to define the integer linear problems required for the calculation of the CRBD bounds provided herein, and perform a quantitative estimation of the impact of the CRBD on a representative application case.

Acknowledgements The authors wish to acknowledge the advice of the anonymous reviewers, which helped improve the initial version of this paper, and the support by Damien Hardy at IRISA/Rennes on the Heptane tool.

[8] The bounds that PIP places on semaphores is not straightforwardly comparable with the bound on PCP.

References

1 S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

2 Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 153–162. ACM, 2010.

3 T. P. Baker. Stack-based Scheduling for Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

4 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *Proc. of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

5 J.V. Busquets-Mataix and A. Wellings. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. of the 2nd Real-time Technology and Application Symposium*, 1996.

6 C. Lee, J. Hahn, Y. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

7 C. G. Lee, K. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transaction on Software Engineering*, 27(9):805–826, 2001.

8 Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.

9 Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206. ACM, 2003.

10 H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks under PIP. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, 2009.

11 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

12 J. Staschulat and R. Ernst. Scalable precision cache analysis for preemptive scheduling. In *Proc. of the 2005 ACM SIGPLAN/SIGBED conference on Languages*, 2005.

13 Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proc. of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

14 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, G. Staschulat, and P. Stenströem. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.

15 V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs, 2004.

Toward Precise PLRU Cache Analysis*

Daniel Grund[1] and Jan Reineke[2]

1 **Saarland University, Saarbrücken, Germany.**
`grund@cs.uni-saarland.de`

2 **University of California, Berkeley, USA.**
`reineke@eecs.berkeley.edu`

Abstract

Schedulability analysis for hard real-time systems requires bounds on the execution times of its tasks. To obtain *useful* bounds in the presence of caches, cache analysis is mandatory.
The subject-matter of this article is the static analysis of the tree-based PLRU cache replacement policy (pseudo least-recently used), for which the precision of analyses lags behind those of other policies. We introduce the term *subtree distance*, which is important for the update behavior of PLRU and closely linked to the peculiarity of PLRU that allows cache contents to be evicted in "logarithmic time". Based on an abstraction of subtree distance, we define a must-analysis that is more precise than prior ones by excluding spurious logarithmic-time eviction.

Keywords and phrases Cache Analysis, PLRU Replacement, PLRU Tree

Digital Object Identifier 10.4230/OASIcs.WCET.2010.23

1 Introduction

In hard real-time systems, one needs to derive offline guarantees for the timeliness of reactions. Thereto, one must determine bounds on the worst-case execution time (WCET) of programs [12]. To obtain tight and thus useful bounds on the execution times, timing analyses *must* take into account the cache architecture of the employed processors. However, developing cache analyses—analyses that statically classify memory accesses as cache hits or cache misses—is a challenging problem.

Besides the determination of addresses that are being accessed, cache analysis is concerned with the analysis of the employed replacement policy. Precise and efficient analyses have been developed early on for LRU [3, 11] and more recently also for FIFO [4, 5]. However, there is a third major policy, PLRU (pseudo least-recently used), which is for instance employed in the TriCore 1798 and several PowerPC variants (MPC603e, MPC755, MPC7448). Compared to analyses of LRU or FIFO, no analyses of similar precision exist for PLRU. The best known PLRU analysis was introduced in [6] for associativity 8 and later categorized as an instance of relative-competitiveness-based analyses [8]. For PLRU, such analyses can at most classify $\log_2(k) + 1$ out of k cached memory blocks as hits.

* The research leading to these results has received funding from or was supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no 216008 (Predator), the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 (AVACS), and the Center for Hybrid and Embedded Software Systems at UC Berkeley (CHESS), the latter of which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 23–35
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In Section 3, we describe three properties of PLRU that make its analysis challenging and coin terms for them: *non-trivial logical states*, *logarithmic-time eviction* and *arbitrary survival.* In Section 4, we address the first one: we adapt knowledge about PLRU [9] and show how to represent the logical state of PLRU cache sets by abstracting from cache set states that are physically different but exhibit the same replacement behavior. Section 5, presents our main contributions. We identify two new sizes of PLRU that are relevant to logarithmic-time eviction: *number of leading zeros* and *subtree distance.* Subsequently, we define a must-analysis that is based on abstractions of those two sizes and can exclude spurious logarithmic-time eviction.

In Section 6 we cover closely-related work including relative-competitiveness-based analyses, against which we compare in Section 7. The introduced analysis is more precise than prior ones and has strong advantages in the analysis of loops, at the cost of an acceptable loss in analysis performance.

2 Foundations

Memory Blocks, Caches, and Access Sequences

Caches store a subset of the main memory's contents to bridge the latency gap between CPU and main memory. To reduce management overhead, main memory is logically partitioned into a set of equally-sized *memory blocks* $\mathcal{B}$. Blocks are cached as a whole in cache lines of equal size. To enable an efficient cache look-up of blocks, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets* $\mathcal{Q}_{P_k}$. The size of a cache set is called the *associativity* k of the cache. As the cache is smaller than the main memory, the number of memory blocks that map to a particular cache set is greater than the size of the cache set. Upon cache misses, a *replacement policy* must decide which memory block to replace. Well-known policies for individual cache sets are least-recently used (LRU), first-in first-out (FIFO), and pseudo-LRU (PLRU) a cost-efficient variant of LRU. For an introduction to caches refer to [7]. A cache set can be formalized by:

- Its domain $\mathcal{Q}_{P_k}$, where the subscript denotes policy and associativity. E.g., $\mathcal{Q}_{\mathrm{FIFO}_k}$ is the set of all FIFO-controlled cache sets of associativity k.
- An update function $U_{P_k} : \mathcal{Q}_{P_k} \times \mathcal{B} \to \mathcal{Q}_{P_k}$, which computes the state of a cache set $q \in \mathcal{Q}_{P_k}$ after a memory block $b \in \mathcal{B}$ has been accessed.

Let $\mathcal{S} := \mathcal{B}^*$ be the set of finite access sequences to memory blocks, e.g. $s_1 := \langle a, b, a, c \rangle$. The update function U_{P_k} can be lifted from a single access to access sequences in the expected way.

Static Analyis

Our work is based on static analysis by abstract interpretation, which abstracts from the concrete program semantics and its respective concrete domain D. Instead, it represents more abstract information in an abstract domain A. The relation between D and A can be given by an abstraction function $\alpha_A : D \to A$ and a concretization function $\gamma_A : A \to D$. For safety properties, one often abstracts from a *collecting semantics.* In that case, D is a powerset domain.

A program is analyzed by performing a fixed-point computation on a set of equations induced by that program. The equations are set up with the help of an *abstract transformer*, $U_A : A \times I \to A$, that describes how abstract values before and after instructions I are

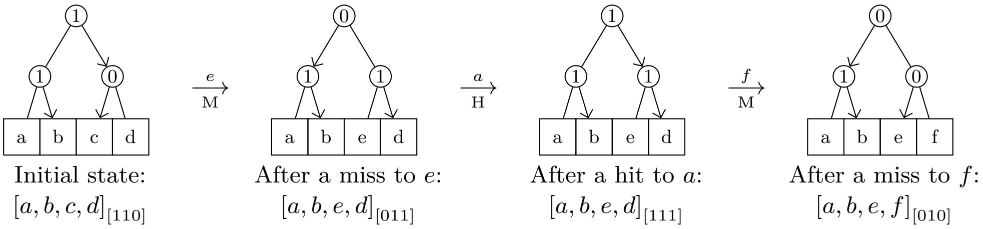


Figure 1 Updates of a PLRU cache set for the access sequence $\langle e, a, f\rangle$.

correlated. If an instruction has multiple predecessors, a *join function* $J_A : A \times A \to A$ combines all incoming values into a single one. For an introduction to abstract interpretation refer to [2].

Static Cache Analysis

The aim of static cache analysis is to classify individual memory accesses as hits (H) or misses (M). However, for some accesses an analysis might fail to classify them as hits or misses, i.e. they remain unclassified ($\top$). The classification domain is given by $Class := \{\mathrm{H}, \mathrm{M}\}^{\top}$.

Static cache analysis by abstract interpretation computes *may*- and *must*-cache information at program points: may- and must-cache information are used to derive upper and lower approximations, respectively, to the *contents* of all concrete cache states that might occur whenever program execution reaches a program point. Must-cache information is used to derive safe information about cache hits. The more cache hits can be predicted, the better the upper bound on the execution times. May-cache information is used to safely predict cache misses.

As most cache architectures manage their cache sets independently from each other, cache analyses can analyze them independently as well. Thus, we limit ourselves to the analysis of a single cache set. For details on (LRU-)cache analysis refer to [3].

3 PLRU: Semantics and Analysis Challenges

Pseudo-LRU (PLRU) is a tree-based approximation of the LRU policy. It arranges the k cache lines at the leaves of a tree with $k-1$ “tree bits” pointing to the line to be replaced/filled next; a 0 indicating the left subtree, a 1 indicating the right. After every access, all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched.

There are at least two variants of PLRU that differ in their handling of invalid cache lines:

Sequential-fill If there are invalid lines upon a cache miss, the least of them (w.r.t. an ordering) is filled. Only if all lines are valid the tree-bits determine which line to replace.

Tree-fill Regardless of invalid lines, the line to be filled or replaced is always determined by the tree-bits.

In the following, we only consider the tree-fill variant. Examine Figure 1: In the initial state, the tree bits point to the line containing memory block c. We textually represent a PLRU-state by the contents of its cache lines and the pre-order traversal of its tree bits. The initial state in the example is thus written $[a,b,c,d]_{[110]}$. A miss to e evicts the memory block c which was pointed to by the tree bits. To protect e from eviction, all tree bits on the path to the root of the tree are made to point away from it. Similarly, upon the following hit to a, the bits on the path from a to the root of the tree are made to point away from a. Note that they are not necessarily flipped. Another access to a would not change the tree bits at all as they already point away from a. Finally, a miss to f eliminates d from the cache set. So, one can represent PLRU cache sets as a k-tuple of memory blocks and $k-1$ bits:

$$q \in \mathcal{Q}_{\mathrm{PLRU}_k} := \mathcal{B}_{\perp}^{k} \times \mathbb{B}^{k-1} \tag{1}$$

Non-trivial logical cache states

Caches implemented in hardware have to satisfy several contradictory optimization goals. For instance, they should provide a low hit latency at a lower power consumption and implementation cost, i.e., area consumption. To satisfy these goals, a cache implementation cannot arbitrarily rearrange the contents of its cache lines upon every access to reflect an access's effect on its logical state. Instead, as in the implementation of PLRU described above, a small number of additional status bits is maintained and updated upon accesses. Due to these status bits, several physical states of the cache represent the same logical state. For static cache analyses it would be inconvenient and inefficient to distinguish such states, as they exhibit the same observable behavior in terms of hits and misses. *The first step in the design of a cache analysis should therefore be to abstract from physical cache states to logical cache states.*

For caches employing LRU or FIFO it is easy to abstract from the physical positions of memory blocks in cache sets. For LRU one can abstract from physical cache set positions by ordering the memory blocks from most-recently to least-recently used, i.e. by their age [3]. For FIFO one can abstract by ordering the blocks from last-in to first-in, i.e. according to their distance to the FIFO pointer [4]. In Section 4, we introduce a sound and complete abstraction from physical cache positions for PLRU, which is more involved due to its "non-linear" tree structure. This abstraction is a coarsest that is still complete: It distinguishes two concrete states if and only if there are access sequences that will result in a different hit/miss behavior.

Logarithmic-time eviction

Consider a cache set of size k: If LRU is employed, it takes at least k accesses to evict a block that has just been accessed [9]. If PLRU is employed, a block might already be evicted after only $\log_2(k)+1$ accesses [9]. Although this is usually not the case, it is challenging for a must-analysis to prove containedness of more than $\log_2(k)+1$ blocks. In Section 5, we introduce a must-analysis that keeps track of correlations between memory blocks in order to exclude spurious logarithmic-time eviction.

Arbitrary survival

May-analysis of PLRU is also more difficult than may-analysis of LRU or FIFO: as opposed to LRU and FIFO, a block b may still be cached after an arbitrary number of accesses to other memory blocks, even if arbitrarily many different blocks are accessed [1]. This makes it challenging for a may-analysis to prove eviction of blocks. The only may-analysis we currently see would be based on the "evict"-metric introduced in [9]. It would have to observe $e(k) = \frac{k}{2}\log_2(k)+1$ *successive* accesses to *pairwise different* blocks in order to be able to then predict a miss. Such an analysis would likely be of little or no practical use.

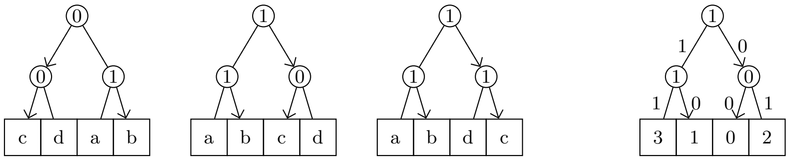


Figure 2 Three equivalent states and a state annotated with edge bits and logical cache positions.

4 Coarsest Complete Abstraction: from Physical to Logical Cache States

Ordering blocks from last-in to first-in in logical states of FIFO and from most- to least-recently used in logical states of LRU is similar. In both cases, blocks are ordered by decreasing miss replacement distance:

▶ **Definition 1** (Miss Replacement Distance)**.** The *miss replacement distance*, $mrd(q, b)$, of a block b is the minimum number of successive misses that evict b from the cache set q. If $b \notin q$, $mrd(q, b) := 0$.

For PLRU, blocks can also be ordered by their miss replacement distance. However, this is more involved. Consider the cache set states in Figure 2. All these states are equivalent with respect to their replacement behavior: if one carries out an arbitrary but fixed access sequence on all states, the same blocks will be evicted in the same order. Given that only misses happen, the blocks will be evicted in the order c, b, d, a. The relation between the physical position of a block and its miss replacement distance is established in four steps:

1. For replacement it does not matter whether a block b is contained in a left or a right subtree. What matters is whether or not a tree bit points to the subtree containing b or not. Hence, we associate an *edge bit* with each edge in the tree. It is 0 if the corresponding tree bit points along this edge and 1 otherwise.
2. Subsequently, we associate an *access path* with each block b in a cache set q. An access path, $ap(q, b)$, is the sequence of edge bits encountered on the path from b to the root of q. If $b \notin q$, then $ap(q, b) = \perp$. In (all of) the above examples, the access path of d is 10 and the one of c is 00.
3. The *logical position* of a block in a cache set is its access path interpreted as a binary number. In the above examples, the logical position of d is 2 and the one of c is 0.
4. The miss replacement distance of a block is its logical position plus one.

For an example, see the rightmost PLRU tree in Figure 2: edges are annotated with edge bits and leaves are annotated with logical cache positions.

▶ **Observation 2** (Access Path Update)**.** *Consider two cached blocks $a \neq b$ with access paths p_a and p_b. Let $p_a = pre_a \circ p_1 \circ suff$ and $p_b = pre_b \circ \overline{p_1} \circ suff$, where $|p_1| = 1$: the paths p_a and p_b start with different prefixes pre_a respectively pre_b, join after the last different bit p_1, and finish with the (possibly empty) shared suffix suff. After accessing b, all tree bits on the path of b point away from b, i.e. all edge bits are 1 and the new access path of b is $p'_b = 1 \ldots 1$. Since a and b share a suffix, setting the tree bits on the path to b also affects a's suffix: its new access path is $p'_a = pre_a \circ \underbrace{0}_{p_1} \circ \underbrace{1 \ldots 1}_{suff}$.*

▶ **Theorem 3** (Miss Replacement Distance [9])**.** *A block b with access path $ap(q, b) = p_1 \ldots p_n$ has miss replacement distance $mrd(q, b) = p_1 \ldots p_n + 1$.*

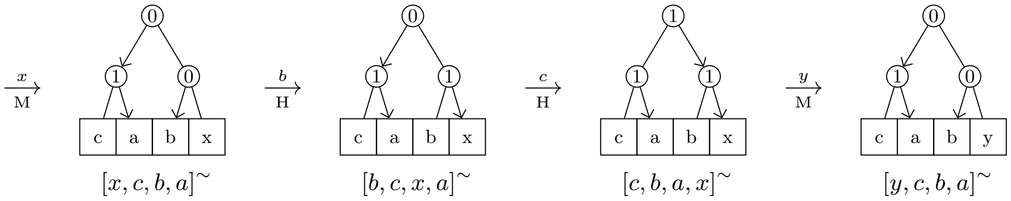


Figure 3 Eviction of x in $\log_2(4) + 1 = 3$ steps by the access sequence $\langle b, c, y \rangle$.

Essentially, the above proceeding defines equivalence classes on PLRU cache sets, i.e. the quotient structure $\mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} := \mathcal{Q}_{\mathrm{PLRU}_k}/\sim$, where the equivalence relation is based on access paths and abstracts from the tree bits. $q_1 \sim q_2 \iff \forall b \in \mathcal{B} : ap(q_1, b) = ap(q_2, b)$. Hence, logical cache set states $\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}}$ can be represented as a function that maps blocks $b \in \mathcal{B}$ to their logical position $\tilde{q}(b)$:

$$\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} = \mathcal{B} \rightarrow \{\perp, 0, \dots, k-1\} \tag{2}$$

In an isomorphic representation as k-tuples of blocks, one can order blocks decreasingly by their logical position (miss replacement distance). For instance, $\tilde{q} = [a, d, b, c]^{\sim}$ represents the three equivalent states in Figure 2: $\tilde{q}(a) = 3, \tilde{q}(d) = 2, \tilde{q}(b) = 1, \tilde{q}(c) = 0$, and $\tilde{q}(x) = \perp$ for all other blocks x.

5 More Precise Must-Analysis Based on Subtree Distances

Logarithmic-time eviction

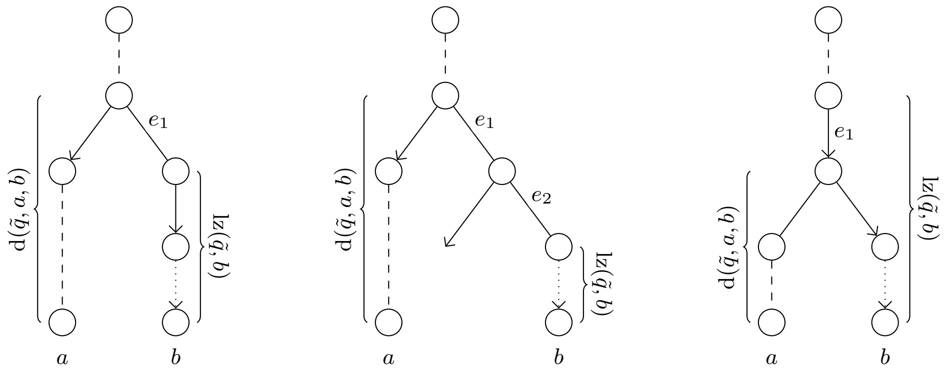
Consider the succession of states in Figure 3. After x has been inserted into the cache set, it only takes 3 accesses to evict it—although the associativity is 4.

Generalized to a k-way PLRU, $\log_2(k) + 1$ is a tight lower bound on the number of accesses that are necessary to evict a just inserted block [9]: After the access to a block x, its access path is $1 \dots 1$. To replace x, all edge bits on its access path must be flipped to $0 \dots 0$. By Observation 2, an access to another block a flips at most one of x's edge bits to 0. Also, the shared suffix of a and x is set to $1 \dots 1$. Hence, to evict x with as few accesses as possible, one set the edge bits to 0 *from left to right* to avoid flipping bits back to 1.

For instance consider Figure 3: The access to b, which is the "direct neighbor" of x, sets the first edge bit of x to 0. The access to c, which is contained in a subtree "that is one step further away" than b, sets the second edge bit of x to 0. Note that accessing a instead of c has the same effect on x. If the cache set was 8-way associative, the third edge bit could be set to 0 by accessing one of the 4 blocks in the "next" subtree. Below we will formally define the notions in double quotes as *subtree distance.*

Sketch of the Analysis

As edge bits in an access path must be set to 0 from left to right to evict a block, the number of leading zeros in access paths is an interesting size. To predict hits, our must-analysis maintains an *upper bound on the number of leading zeros*: as long as this bound is less than $\log_2(k)$ for a block, that block can not be evicted. To improve analysis precision, we additionally maintain *approximations on subtree distances.* With information about subtree distances, the analysis can model the flipping of tree-bits more precisely and is able to exclude more spurious behavior, e.g. the logarithmic-time eviction of blocks.



(a) $\mathrm{d}(\tilde{q}, a, b) = \mathrm{lz}(\tilde{q}, b) + 1$, $e_1 = 1, e_1' = 0$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{lz}(\tilde{q}, b) + 1$

(b) $\mathrm{d}(\tilde{q}, a, b) > \mathrm{lz}(\tilde{q}, b) + 1$, Although $e_1' = 0$: $e_2 = e_2' = 1$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{lz}(\tilde{q}, b)$

(c) $\mathrm{d}(\tilde{q}, a, b) < \mathrm{lz}(\tilde{q}, b) + 1$, $e_1 = 0, e_1' = 1$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{d}(\tilde{q}, a, b)$

Figure 4 Update of the number of leading zeros $\mathrm{lz}(\tilde{q}, b)$ upon a cache hit to block a.

Leading Zeros and Subtree Distance

▶ **Definition 4** (Number of Leading Zeros)**.** The *number of leading zeros* of a block, $\mathrm{lz}(b)$, is the number of leading zeros ($\mathrm{nlz} : \mathbb{N} \to \mathbb{N}$)[1] in the access path of that block ($\tilde{q}(b)$):

$$\mathrm{lz} : \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \times \mathcal{B} \to \{\perp, 0, \ldots, \log_2(k)\} \tag{3}$$

$$\mathrm{lz}(\tilde{q}, b) := \begin{cases} \mathrm{nlz}(\tilde{q}(b)) & : \tilde{q}(b) \neq \perp \\ \perp & : \text{otherwise} \end{cases} \tag{4}$$

For example, in the state $\tilde{q} = [x, c, b, a]^{\sim}$ of Figure 3, $\mathrm{lz}(\tilde{q}, a) = 2, \mathrm{lz}(\tilde{q}, b) = 1, \mathrm{lz}(\tilde{q}, c) = \mathrm{lz}(\tilde{q}, x) = 0$, and $\mathrm{lz}(\tilde{q}, y) = \perp$ for all other blocks y.

▶ **Definition 5** (Subtree Distance)**.** The *subtree distance* between two cached blocks, $\mathrm{d}(\tilde{q}, a, b)$, is the distance to their least common ancestor in the tree.

$$\mathrm{d} : \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \times \mathcal{B} \times \mathcal{B} \to \{\perp, 0, \ldots, \log_2(k)\} \tag{5}$$

$$\mathrm{d}(\tilde{q}, a, b) := \begin{cases} \log_2(k) - \mathrm{ntz}(\tilde{q}(a) \oplus \tilde{q}(b)) & : \tilde{q}(a) \neq \perp, \tilde{q}(b) \neq \perp \\ \perp & : \text{otherwise} \end{cases} \tag{6}$$

If both blocks are cached ($\tilde{q}(\cdot) \neq \perp$) the subtree distance between them is the height of the tree ($\log_2(k)$) minus the length of their shared suffix ($\mathrm{ntz}(\cdot)$). Otherwise, the distance is undefined ($\perp$). Assuming two's complement binary encoding, the length of the shared suffix can be computed by: First, a bitwise `xor` ($\oplus$) of the logical positions, which produces a 0 bit if two bits are equal. Then, the number of trailing zeros[1] in the result is the length of the shared suffix. For example, in the first tree of Figure 3, $\mathrm{d}(\tilde{q}, c, c) = 0, \mathrm{d}(\tilde{q}, c, a) = 1, \mathrm{d}(\tilde{q}, c, b) = \mathrm{d}(\tilde{q}, c, x) = 2$, and $\mathrm{d}(\tilde{q}, a, b) = \mathrm{d}(\tilde{q}, a, x) = 2$, and so on.

To see how the number of leading zeros is updated upon a cache hit to a block a, consider Figure 4. In the successor state $\tilde{q}'$ of $\tilde{q}$, the number of leading zeros after a hit to a ($\tilde{q}(a) \neq \perp$)

[1] $\mathrm{nlz}, \mathrm{ntz} : \mathbb{N} \to \mathbb{N}$ compute the number of leading/trailing zeros of two's-complement numbers. For definitions and efficient implementations see [10].

is

$$\mathrm{lz}(\tilde{q}', b) = \begin{cases} \bot & : \mathrm{lz}(\tilde{q}, b) = \bot \\ \mathrm{lz}(\tilde{q}, b) + 1 & : \mathrm{d}(\tilde{q}, b, a) = \mathrm{lz}(\tilde{q}, b) + 1 \qquad \text{Figure 4a} \\ \min\{\mathrm{lz}(\tilde{q}, b), \mathrm{d}(\tilde{q}, b, a)\} & : \text{otherwise} \qquad \text{Figures 4b, 4c} \end{cases} \tag{7}$$

Upon a cache miss to block a ($\tilde{q}(a) = \bot$) the logical position of each cached block is decremented ($\tilde{q}(b) - 1$), see Definition 1 and Theorem 3. Furthermore, the block at logical position 0 is evicted and trivially $\mathrm{lz}(\tilde{q}', a) = 0$:

$$\mathrm{lz}(\tilde{q}', b) = \begin{cases} 0 & : b = a \\ \mathrm{lz}(\tilde{q}(b) - 1) & : b \neq a, \tilde{q}(b) \neq \bot, \tilde{q}(b) > 0 \\ \bot & : \text{otherwise} \end{cases} \tag{8}$$

Abstraction

As explained above, the number of leading zeros is decisive for the question whether a block can be evicted. Hence, the first constituent of our abstract domain is the number of *potential leading zeros*:

$$\mathit{plz} \in \mathit{PLZ}_k := \mathcal{B} \to \{0, \dots, \log_2(k), \top\} \tag{9}$$

As the prefix *potential* suggests, $\mathit{plz}(b)$ is an upper bound on the number of leading zeros in the access path of b. If $\mathit{plz}(b) = \top$, then b may not be cached (anymore). Formally, the meaning of $\mathit{plz} \in \mathit{PLZ}_k$ is given by the concretization function:

$$\gamma_{\mathit{PLZ}_k} : \mathit{PLZ}_k \to \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \tag{10}$$

$$\gamma_{\mathit{PLZ}_k}(\mathit{plz}) := \{\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \mid \mathit{plz}(b) \neq \top \Rightarrow 0 \leq \mathrm{lz}(\tilde{q}, b) \leq \mathit{plz}(b)\} \tag{11}$$

For the analysis to be able to exclude the possibility of logarithmic-time eviction, it needs information about subtree distances. To see this, consider Equation 7, specifically that the second case depends on $\mathrm{d}(\tilde{q}, b, a)$: If the analysis had *no* information about the subtree distance $\mathrm{d}(\tilde{q}, b, a)$, it would have to conservatively take into account the case that $\mathrm{d}(\tilde{q}, b, a) = \mathrm{lz}(\tilde{q}, b) + 1$. This would mean that upon an access to block a, the upper bound $\mathit{plz}(b)$ would have to be incremented for all $b \neq a$. Ultimately, *an analysis that abstracts completely from subtree distances can never exclude logarithmic-time eviction.* Consequently, to increase analysis precision, the second constituent of our abstract domain maintains some information about subtree distances.

There are several ways to approximate subtree distances in an abstract domain. We chose an abstraction such that abstract elements can be represented efficiently. For each pair of blocks we distinguish between four classes of distances; zero, non-maximal, maximal, and unknown:

$$\mathit{AD}_k := \mathcal{B} \times \mathcal{B} \to \{\{0\}, [1, \log_2(k)), \{\log_2(k)\}, \top\} \tag{12}$$

Formally, the meaning of $\mathit{ad} \in \mathit{AD}_k$ is given by:

$$\gamma_{\mathit{AD}_k} : \mathit{AD}_k \to \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \tag{13}$$

$$\gamma_{\mathit{AD}_k}(\mathit{ad}) := \{\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \mid \mathit{ad}(a, b) \neq \top \Rightarrow \mathrm{d}(\tilde{q}, a, b) \in \mathit{ad}(a, b)\} \tag{14}$$

Although the domain $\mathcal{B} \times \mathcal{B}$ of an $ad \in AD_k$ is of size $O(|\mathcal{B}|^2)$, each $ad \in AD_k$ can be stored in size $O(|\mathcal{B}|)$ by using a set of two disjoint sets of blocks $\{B_0, B_1\}$:

$$\begin{aligned} ad(a,b) &= \{0\} && \iff a = b, \exists i : a \in B_i \\ ad(a,b) &= [1, \log_2(k)) && \iff a \neq b, \exists i : a, b \in B_i \\ ad(a,b) &= \{\log_2(k)\} && \iff a \neq b, \exists i : a \in B_i, b \in B_{1-i} \\ ad(a,b) &= \top && \iff \{a,b\} \not\subseteq B_0 \cup B_1 \end{aligned} \tag{15}$$

Blocks with maximal distance are contained in different sets, blocks with non-maximal distance are contained in the same set. For instance $(\{a,b\},\{c\})$ corresponds to $ad(a,b) = [1, \log_2(k))$, $ad(a,c) = ad(b,c) = \{\log_2(k)\}$, and $ad(x,y) = \top$ for all other $x \neq y$.

Now that PLZ_k and AD_k are introduced, we define the abstract domain of the must-analysis, which is a partial function that associates bounds on leading zeros with approximations of subtree distances:

$$Plru_k^{AD} := AD_k \hookrightarrow PLZ_k \tag{16}$$

The set of concrete cache set states represented by $\hat{q} \in Plru_k^{AD}$ is determined by:

$$\gamma^{AD}_{Plru_k} : Plru_k^{AD} \to \mathcal{P}(\mathcal{Q}_{\mathrm{PLRU}_k^{\sim}}) \tag{17}$$

$$\gamma^{AD}_{Plru_k}(\hat{q}) := \bigcup_{ad \in dom(\hat{q})} \gamma_{AD_k}(ad) \cap \gamma_{PLZ_k}(\hat{q}(ad)) \tag{18}$$

A concrete state must satisfy any of the distance constraints $\gamma_{AD_k}(ad)$ and the associated constraints on leading zeros $\gamma_{PLZ_k}(\hat{q}(ad))$.

Classification

An access to a block b can be classified as a hit if its number of leading zeros is at most $\log_2(k)$ for all approximations of subtree distances. Otherwise, the access might be a miss.

$$C^{AD}_{Plru_k} : Plru_k^{AD} \times \mathcal{B} \to Class \tag{19}$$

$$C^{AD}_{Plru_k}(\hat{q}, b) := \begin{cases} \mathrm{H} & : \forall ad \in dom(\hat{q}) : \hat{q}(ad)(b) \neq \top \\ \top & : \text{otherwise} \end{cases} \tag{20}$$

Join

For coinciding approximations of subtree distances, the associated approximation of leading zeros is joined by taking the maximum bound for each block:

$$J^{AD}_{Plru_k}(\hat{q}_1, \hat{q}_2) := \lambda ad. \begin{cases} \hat{q}_1(ad) & : ad \in dom(\hat{q}_1) \setminus dom(\hat{q}_2) \\ \hat{q}_2(ad) & : ad \in dom(\hat{q}_2) \setminus dom(\hat{q}_1) \\ J_{PLZ_k}(\hat{q}_1(ad), \hat{q}_2(ad)) & : ad \in dom(\hat{q}_1) \cap dom(\hat{q}_2) \end{cases} \tag{21}$$

$$J_{PLZ_k}(plz_1, plz_2) := \begin{cases} \lambda b. \max\{plz_1(b), plz_2(b)\} & : plz_1(b) \neq \top, plz_2(b) \neq \top \\ \top & : \text{otherwise} \end{cases} \tag{22}$$

Update

The update of $Plru_k^{AD}$ is based on the updates of AD_k and PLZ_k. First, consider $U_{AD_k} : AD_k \times PLZ_k \times \mathcal{B} \times Class \to 2^{AD_k}$:

$$U_{AD_k}(\{B_0, B_1\}, plz, a, \mathrm{H}) := \{\{B_0', B_1\} \mid B_0' := B_0 \cup \{a\}, a \notin B_1, |B_0'| \leq k/2\} \tag{23}$$

$$U_{AD_k}(\{B_0, B_1\}, plz, a, \mathrm{M}) := \{\{B_0', B_1\} \mid B_0' := B_0 \backslash \{x\} \cup \{a\}, plz(x) \in \{\log_2(k), \top\}, |B_0'| \leq k/2\} \tag{24}$$

$$U_{AD_k}(ad, plz, a, \top) := U_{AD_k}(ad, plz, a, \mathrm{H}) \cup U_{AD_k}(ad, plz, a, \mathrm{M}) \tag{25}$$

Upon a hit to block a, a must be cached. Hence, we can make assumptions about its distances by adding it to a set B_i. To maintain consistency, a must not be contained in both sets simultaneously ($a \notin B_{1-i}$). Furthermore, at most $k/2$ blocks can have non-maximal subtree disctance to each other ($|B_i'| \leq k/2$). (Note that B_0 and B_1 are interchangeable since $\{B_0, B_1\}$ is a set.)

Upon a cache miss, the accessed block a inherits its subtree distances from the replaced block x. Due to the abstraction, several blocks might come into consideration for eviction, namely all blocks with $plz(x) \in \{\log_2(k), \top\}$.

If the access is unclassified, one has to take the union of the results of the hit- and miss-update.

The update of the potential leading zeros closely resembles the three cases in Figure 4:

$$U_{PLZ_k}(plz, ad, a) := \lambda b. \begin{cases} 0 & \text{if } a = b \\ \top & \text{else if } ad(a, b) = \top \\ plz(b) & \text{else if } plz(b) + 1 < L \\ plz(b) + 1 & \text{else if } L \leq plz(b) + 1 \leq U \\ U & \text{else if } plz(b) + 1 > U \end{cases} \tag{26}$$

Since the subtree distances are approximated, one has to rely on lower and upper bounds ($L \equiv \min\{n \in ad(a, b)\}, U \equiv \max\{n \in ad(a, b)\}$) of the interval $ad(a, b) \neq \top$. Consider the fourth case for instance: since $plz(b) + 1 = \mathrm{d}(\tilde{q}, a, b)$ *might* be possible, one has to increment $plz(b)$.

The update on $Plru_k^{AD}$ assigns each subtree distance approximation ad' an updated approximation of leading zeros. Different approximations ad might be updated to the same ad'. Hence, one must join ($\sqcup$) all updated approximations of leading zeros ($U_{PLZ_k}()$) of all ad for which $ad' \in U_{AD_k}(ad, \ldots)$.

$$U_{Plru_k}^{AD}(\hat{q}, a, cl) := \\ \lambda ad'. \bigsqcup \{U_{PLZ_k}(\hat{q}(ad), ad', a) \mid ad \in dom(\hat{q}), ad' \in U_{AD_k}(ad, \hat{q}(ad), a, cl)\} \tag{27}$$

Uncertainty about accessed addresses

Cache analysis comprises *value analysis* and *replacement analysis.* Value analysis determines approximations to accessed addresses, which are the inputs to the cache. Given the accessed addresses, replacement analysis determines approximations to cache contents. Therefore a replacement analysis is generally applicable to instruction, data, and unified caches.

There are cases where the value analysis cannot precisely determine the address of a memory access. Nonetheless, the replacement analysis can *always* handle such uncertainty in a *sound* way: a sound successor state can be computed by performing updates of the current state for all addresses that might be accessed and then joining all those states into a single one. However, this way, the uncertainty about accessed addresses translates into additional uncertainty about cache set states, which might translate into less classified accesses.

6 Closely Related Work

The related works most relevant for this paper are cache analyses of LRU [3], FIFO [4, 5] and PLRU [8]. [3] introduces the concepts of may- and must-caches and present may- and must-analyses for LRU that are based on abstract interpretation. [4, 5] introduce several must- and a may-analyses for FIFO and show how to combine the corresponding abstract domains in order to improve analysis precision. For pointers to earlier work on cache analysis directed at WCET analysis and other cache analyses, we kindly refer the reader to [4].

The only prior analysis of PLRU, $Plru_k^{RC}$, is a must-analysis based on relative competitiveness [8]. Under certain conditions, relative competitiveness allows one to use cache analyses for one policy as cache analyses for other policies. For instance, an LRU must-analysis for a $\log_2(k) + 1$-way associative cache can be employed as a must-analysis for a k-way PLRU.

7 Evaluation

In the following, we compare to each other
(a) the analysis $Plru_k^{AD}$ presented in this paper,
(b) the analysis $Plru_k^{RC}$ based on relative competitiveness [8] as explained in Section 6, and
(c) the collecting semantics $Plru_k^{CS}$ of PLRU.
The collecting semantics is the exact set of cache set states that may reach a program point. It delimits the precision of *any* static analysis. If a memory access cannot be classified as hit or miss in the collecting semantics, no sound static analysis can do so. We computed it using an analysis based on a powerset domain of symbolically-represented concrete cache-set states.

To quantify the precision of the analyses, we applied the analyses to two parametrizable classes of synthetic benchmarks, where the parameter n controls the level of temporal locality: *Loop(n)* is a loop that iterates 16 times and accesses n different blocks, i.e. $(1\ 2 \ldots n)^{16}$. *Rand(n)* is a set of 100 sequences, each containing 100 randomly distributed accesses to n different blocks, i.e. $(1|2|\ldots|n)^{100}$.

The results for associativities $k = 4$ and $k = 8$ are shown in Table 1 (for $k = 2$, PLRU is identical to LRU). Except for one negligible exception, $Plru_k^{AD}$ can guarantee higher hit

Table 1 Guaranteed hit rates [%] provided by the two analyses and the collecting semantics.

		Associativity $k = 4$				Associativity $k = 8$						
	n	2	3	4	5	2	3	4	5	6	7	8
Loop	$Plru_k^{RC}$	93.8	93.8	0.0	0.0	93.8	93.8	93.8	0.0	0.0	0.0	0.0
	$Plru_k^{AD}$	93.8	93.8	92.2	0.0	93.8	93.8	93.8	92.5	90.6	0.0	0.0
	$Plru_k^{CS}$	93.8	93.8	92.2	0.0	93.8	93.8	93.8	92.5	91.7	90.2	86.7
Rand	$Plru_k^{RC}$	98.0	97.0	73.1	58.8	98.0	97.0	96.0	77.7	64.4	55.7	48.1
	$Plru_k^{AD}$	98.0	97.0	94.7	75.4	98.0	97.0	95.8	93.0	84.3	63.5	52.0
	$Plru_k^{CS}$	98.0	97.0	94.7	75.4	98.0	97.0	96.0	93.9	91.0	84.0	68.4

rates than $Plru_k^{RC}$.

For the *Loop* benchmarks, $Plru_k^{RC}$ cannot classify any hits if $n > \log_2(k) + 1$, wheras $Plru_k^{AD}$ can do so for up to $n = 2\log_2(k)$. Hence, $Plru_k^{AD}$ is preferable for loops containing more than $\log_2(k) + 1$ different accesses. If the analyses predict hits, the amount is close to the limit given by $Plru_k^{CS}$.

The *Rand* benchmarks are stress tests for the abstract domains. For smaller n, both analyses perfom equally well. For $n > \log_2(k) + 1$, the gap between the collecting semantics and the analysis results grows. With increasing n, $Plru_k^{RC}$ falls behind $Plru_k^{AD}$.

Regarding the efficiency of the analyses, please note that $Plru_k^{AD}$ and $Plru_k^{CS}$ are implemented as prototypes whereas $Plru_k^{RC}$ is tuned. For each analysis we measured the overall time needed to complete all benchmarks. For $k = 4$, all analyses took around 3.3s. For $k = 8$, $Plru_k^{RC}$ completed after 3.5s, $Plru_k^{AD}$ after 5.5s, and $Plru_k^{CS}$ after 12.5s. Compared to $Plru_k^{RC}$, the disadvantage of $Plru_k^{AD}$ is that it is a disjunctive domain, which entails higher memory consumption and lower performance.

8 Conclusions and Further Work

Our first contribution, the $Plru_k^{AD}$ analysis, has pros and cons: It is more precise than its sole competitor $Plru_k^{RC}$. Most importantly, it can classify hits in “larger” loops, where $Plru_k^{RC}$ cannot. On the other hand, its higher memory consumption might hamper scalability. However, tradeoffs are possible by changing the approximation of subtree distances, i.e. plugging-in different domains AD_k.

Our second contribution, the understanding of the *subtree distance* and its relation to other sizes, is perhaps more valuable than the analysis itself: For FIFO, it was shown to be beneficial for precision to refine the abstract transformer by discriminating between hits and misses [4]. For PLRU, this is not sufficient as a hit can both, accelerate or defer the eviction of other blocks. Instead, we consider the subtree distance as an important size in the design of future PLRU (may-)analyses, which will possibly degrade $Plru_k^{AD}$ to a proof-of-concept analysis.

References

1 Christoph Berg. PLRU cache domino effects. In *WCET*, 2006.

2 Patrick Cousot and Radhia Cousot. *Building the Information Society*, chapter Basic Concepts of Abstract Interpretation, pages 359–366. Kluwer Academic Publishers, 2004.

3 Christian Ferdinand. *Cache Behaviour Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

4 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 120–136. Springer-Verlag, August 2009.

5 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *ECRTS*, 2010.

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

7 John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan Kaufmann, 2003.

8 Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *LCTES ’08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on*

Languages, Compilers, and Tools for Embedded Systems, pages 51–60, New York, NY, USA, 2008. ACM Press.

9 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

10 Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

11 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 *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, page 192, Washington, DC, USA, 1997. IEEE Computer Society.

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

Integrating Abstract Caches with Symbolic Pipeline Analysis

Stephan Wilhelm[1] **and Christoph Cullmann**[1]

1 **AbsInt Angewandte Informatik GmbH, Science Park 1; D-66123 Saarbrücken, Germany**

Abstract

Static worst-case execution time analysis of real-time tasks is based on abstract models that capture the timing behavior of the processor on which the tasks run. For complex processors, task-level execution time bounds are obtained by a state space exploration which involves the abstract model and the program. Partial state space exploration is not sound. Symbolic methods using binary decision diagrams (BDDs) allow for a full state space exploration of the pipeline, thereby maintaining soundness. Caches are too large to admit an efficient BDD representation. On the other hand, invariants of the cache state can be computed efficiently using abstract interpretation. How to integrate abstract caches with symbolic-state pipeline analysis is an open question [11]. We propose a semi-symbolic domain to solve this problem. Statistical data from industrial-level software and WCET tools indicate that this new domain will enable an efficient analysis.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.36

1 Introduction

The execution time of a task depends on the execution speed of the processor on which the task runs, as well as on the executed program code and on input values. Further, complex processors implement various features to reduce the average execution time, e.g., pipelines and caches. Execution times on such processors also depend on the execution history and on the start state of the hardware [7, 9]. As a consequence, tools for safe WCET prediction have to cover all feasible program paths, inputs, and hardware states.

Static WCET analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior [5]. However, abstraction loses information which leads to uncertainty, e.g., it may not be possible to statically determine the exact address of a memory access. Furthermore, program inputs are not precisely known in advance. At the level of the hardware model, this lack of information is accounted for by non-deterministic choices. To be safe, the analysis has to explore all possibilities. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints [10].

In [13] we presented a symbolic approach for pipeline analysis that avoids the explicit enumeration of reachable pipeline states, and showed its effectiveness in alleviating the state explosion problem in WCET analysis. The implementation cooperates efficiently with a framework of static analyses based on abstract interpretation. A commonality of these analyses is the fact that they run prior to pipeline analysis. Hence, cooperation boils down to importing statically available analysis results. In contrast, the abstract interpretation of caches [6] cannot be separated from pipeline analysis. The cache state depends on the order of memory accesses and therefore on the state of the pipeline. The pipeline state in turn is influenced by the latency of instruction and data fetches which depends on the cache state. Explicit-state implementations of pipeline analysis establish a one-to-one relationship

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 36–43
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

between pipeline and cache states, i.e., they combine each abstract cache state with a single abstract pipeline state. The pipeline state triggers an update of its associated cache state whenever the processor accesses a cached memory area.

Symbolic-state implementations cannot afford a one-to-one relation between pipeline and cache states without losing the advantages of symbolic state space exploration. We present experimental evidence that a one-to-one relation between pipeline and cache states is not required in practically relevant scenarios. Furthermore, we describe a semi-symbolic domain that efficiently integrates abstract interpretation based cache analysis with symbolic pipeline analysis while preserving a high analysis precision.

2 The Problem

Notation. The sets of natural numbers and Boolean values are denoted by $\mathbb{N}$ and $\mathbb{B}$, respectively. $\mathbb{I} \subset \mathbb{N} \times \mathbb{N}$ is the set of intervals such that $\forall (l,u) \in \mathbb{I} : l \leq u$. We write $f \cdot g$ for conjunction, $f + g$ for disjunction, and $\overline{f}$ for negation of Boolean functions and variables. A vector of Boolean values is written as $\vec{x}$.

Pipeline analysis [10, 4] computes upper bounds on the execution time of basic blocks using an abstract pipeline model. The model accounts for timing-relevant processor components, such as pipelining, speculation, and peripheral hardware. To reduce complexity, ALUs and register files are handled by a dedicated value analysis [3]. *Symbolic* pipeline analysis [13] uses BDD representations [2] of abstract pipeline models and sets of abstract states. The involved BDDs are directed, acyclic graphs that represent Boolean functions of type $\mathbb{B}^n \to \mathbb{B}$. An example BDD for the Boolean function $x_0 \cdot \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} \cdot \overline{x_5} + \overline{x_0} \cdot (x_1 + \overline{x_1} \cdot x_2)$ is depicted in Fig. 2.

The idea behind the symbolic approach is, that an abstract pipeline model corresponds to a finite state machine (FSM) with n Boolean state variables. Assignments of the state variables define pipeline states, e.g., in terms of different positions of instructions (identified by their addresses) in the pipeline. An FSM consists of a set of states $Q \subseteq \mathbb{B}^n$, a set of initial states $S \subseteq Q$ and a transition relation $T \subseteq Q \times Q$. Each set of states $A \subseteq Q$, as well as the transition relation T, can be associated with a Boolean function $\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$ where $\mathbf{A}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in A$ and $\mathbf{T} : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$ where $\mathbf{T}(\vec{x}, \vec{y}) = 1 \Leftrightarrow (\vec{x}, \vec{y}) \in T$. We say that $\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$ is the *characteristic function* of the set A. The pipeline model is given in terms of its symbolic transition relation by the BDD $\mathbf{T}_{\mathcal{M}}$. Static program information, such as branch targets and intervals of register contents, are encoded into a BDD program relation $\mathbf{T}_{\mathcal{L}}$ that restricts the possible transitions of $\mathbf{T}_{\mathcal{M}}$. A set of pipeline states is represented by a BDD $\mathbf{A}$. State traversal is implemented by repeated application of a symbolic image operator $\mathbf{Img} : (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}) \times (\mathbb{B}^n \to \mathbb{B}) \to (\mathbb{B}^n \to \mathbb{B})$ [8]. The set of successor states for the states in $\mathbf{A}$ is computed by $\mathbf{Img}(\mathbf{T}_{\mathcal{M}} \cdot \mathbf{T}_{\mathcal{L}}, \mathbf{A})$.

Cache analysis [6] operates on abstract representations of cache states. The abstract representation allows to trade precision for efficiency. Soundness is maintained by losing information only on the safe side, i.e., the result over-approximates the concrete cache states but it never misses a reachable cache state. The interface of the cache analysis comprises functions to query and update abstract caches with intervals of memory addresses. It also features a join operator for joining two cache states into another cache state that over-approximates both. The join operation may lose precision. There are two possibilities for integrating a symbolic-state implementation of pipeline analysis with a cache representation:

1. Including the cache into the symbolic representation of pipeline states.

2. Associating an abstract cache representation with a symbolic representation of pipeline states.

Let us consider the first approach. Even small caches are too large to admit a straightforward BDD representation for symbolic state traversal. In [12] we proposed an alternative symbolic representation for caches. Compactness was achieved by losing the correlation between the abstract cache cells; the resulting BDD is no longer the characteristic function of a set of hardware states. Unfortunately, it seems that – despite its compactness – the proposed representation does not allow for an efficient state traversal. So far, attempts to design efficient image operators, i.e., operators that avoid an exhaustive enumeration of the encoded states, have not been successful.

The second possibility seems equivalent to the approach taken by explicit-state implementations. However, symbolic-state implementations cannot afford a one-to-one relation between pipeline and cache states without losing the advantages of symbolic state space exploration. The explicit handling of caches would require the same explicit enumeration of pipeline states that the symbolic representation is trying to avoid. The next section presents a domain that is based on this second possibility, but uses a more favorable relation between pipeline and cache states.

3 Proposed Domain

We propose a semi-symbolic domain that integrates an abstract cache representation with a symbolic representation of pipeline states. The explicit enumeration problem is avoided by maintaining an efficient relation between pipeline and cache states. The basic idea is that we combine a set of pipeline states (represented symbolically by a BDD) with a single abstract cache state. The product of the pipeline and cache domains is thus based on an *n-to-one* relation. This allows us to preserve the benefits of the symbolic representation by manipulating sets of pipeline states symbolically.

Let $\widehat{\mathcal{C}}$ and $\mathbb{B}^n \to \mathbb{B}$ denote the abstract cache domain and the symbolic pipeline domain, respectively. A partition of abstract hardware states is a tuple of type $(\mathbb{B}^n \to \mathbb{B}) \times \widehat{\mathcal{C}}$ and $\mathcal{H}$ denotes the set of all partitions. The proposed domain $\mathcal{D}$ is the power set of $\mathcal{H}$ excluding the empty set.

3.1 Updating partitions of abstract hardware states

We show the update of a single partition $(\mathbf{A}, \widehat{a}) \in \mathcal{H}$, where $\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$ is a BDD representing a set of pipeline states and $\widehat{a} \in \widehat{\mathcal{C}}$ is an abstract cache state. Let $\mathcal{A}_C$ be the set of all addresses in cached memory that are accessed by the analyzed program. For the remainder of this paper we assume that *all* memory accesses address cached memory regions. The pipeline model then needs $m = log_2(|\mathcal{A}_C|)$ state variables for addressing memory. We require that these variables appear first in the BDD representation. The addressed interval can be obtained by a function $acc : (\mathbb{B}^n \to \mathbb{B}) \to \mathbb{I}$ that inspects the first m BDD variables. Its implementation is discussed in Sec. 3.3.

The classification function $cl : \widehat{\mathcal{C}} \times \mathbb{I} \to \{(0,1),(1,0),(1,1)\}$ of the abstract cache domain determines whether an access results in a cache hit $(0,1)$ or miss $(1,0)$. Note that the result of this query can also be *undecided* $(1,1)$ if precise information has been lost due to abstraction or if the interval comprises both, cache hits and misses. The result of $cl(\widehat{a}, acc(\mathbf{A}))$ can be encoded as a symbolic relation $\mathbf{T}_C$ by a function $enc : \mathbb{B}^2 \to (\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B})$. The computed relation restricts the possible transitions of the model relation $\mathbf{T}_{\mathcal{M}}$. It only allows

```
$step(\mathbf{A}, \widehat{a}) =$
    let $I = acc(\mathbf{A})$ in
    let $\mathbf{T}_C = enc(cl(\widehat{a}, I))$ in
        $(\mathbf{Img}(\mathbf{T}_\mathcal{M} \cdot \mathbf{T}_\mathcal{L} \cdot \mathbf{T}_C, \mathbf{A}), up(\widehat{a}, I))$
```

Figure 1 Implementation of the update function $step : \mathcal{H} \rightarrow \mathcal{H}$.

for transitions that correspond to the result of the cache query. This is analogue to the construction of $\mathbf{T}_\mathcal{L}$ from statically available program information.

Let $up : \widehat{\mathcal{C}} \times \mathbb{I} \rightarrow \widehat{\mathcal{C}}$ denote the update function for abstract cache states. The update of a single partition $(\mathbf{A}, \widehat{a})$ can then be computed by a function $step : \mathcal{H} \rightarrow \mathcal{H}$ as depicted in Fig. 1. The *step* function first determines the interval I of memory addresses that is accessed by the pipeline states in $\mathbf{A}$. It then queries the cache domain to determine whether the access hits or misses the cache and – based on this information – constructs the BDD $\mathbf{T}_C$ for restricting the reachable pipeline states. The constructed BDD is conjoined with the BDDs $\mathbf{T}_\mathcal{M}$ and $\mathbf{T}_\mathcal{L}$ to obtain the effective transition relation for the next update. By application of the image operator on the computed transition relation and the set of pipeline states $\mathbf{A}$, it computes the set of successor pipeline states. The next cache state is obtained by application of the cache domain update function on the current cache $\widehat{a}$ and the accessed interval I.

3.2 Balancing pipeline and cache states

In order to maintain a favorable n-to-one relation between pipeline and cache states, we introduce a balancing operation to be applied in each round of the state traversal. The balancing operation involves two steps: partition and join. The partition step is based on the decomposition of the BDD of pipeline states. Decomposition of a BDD $f : \mathbb{B}^n \rightarrow \mathbb{B}$ into its cofactors with respect to a variable x_n means computing subfunctions $g, h : \mathbb{B}^{n-1} \rightarrow \mathbb{B}$ such that $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, 0)$ and $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, 1)$. This decomposition, also known as Shannon expansion, is a very efficient operation on BDDs and the foundation of many basic BDD algorithms.

Let $(\mathbf{A}, \widehat{a}) \in \mathcal{H}$ be a partition of abstract hardware states. The first m state variables in $\mathbf{A}$ encode the accessed interval of memory addresses. We partition $(\mathbf{A}, \widehat{a})$ by a function $part : \mathcal{H} \rightarrow \mathcal{D}$ that recursively decomposes $\mathbf{A}$ into its cofactors with respect to the first m state variables. A new partition is created for each cofactor together with a copy of the cache state $\widehat{a}$. The decomposition proceeds until all satisfying paths of the BDD pass through the variable $m + 1$. As a result, all pipeline states in a new partition $(\mathbf{A}_x, \widehat{a}) \in part(\mathbf{A}, \widehat{a})$ access the same interval of memory addresses. Note that $step(\mathbf{A}_x, \widehat{a})$ yields a more precise successor cache state than $step(\mathbf{A}, \widehat{a})$.

Excessive partitioning might lead us back to the explicit enumeration problem. In the worst case, each partition in a domain element $D \in \mathcal{D}$ encodes only a single pipeline state. We prevent this by applying a join operator to partitions of D. Let $\sqcup$ denote the join operator for abstract caches [6]. The union of two sets of pipeline states is implemented by disjunction of their characteristic functions. Two partitions $(\mathbf{A}, \widehat{a})$ and $(\mathbf{B}, \widehat{b})$ are joined by a function $join : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$:

$$join((\mathbf{A}, \widehat{a}), (\mathbf{B}, \widehat{b})) = (\mathbf{A} + \mathbf{B}, \widehat{a} \sqcup \widehat{b})$$

To minimize the loss of cache precision, we join only partitions whose pipeline states access the same interval of memory addresses. This restriction also prevents us from undoing the partitioning. The loss in cache precision could be limited further by joining only hardware states with similar caches. This however requires a similarity metric for abstract cache states. A simple but efficient similarity metric would be, to only join two cache states $\widehat{a}, \widehat{b} \in \widehat{\mathcal{C}}$ if one of them already over-approximates the other, which is equivalent to

$$\widehat{a} \sqcup \widehat{b} = \widehat{a} \quad \text{or} \quad \widehat{a} \sqcup \widehat{b} = \widehat{b}$$

Besides balancing the relation between pipeline and cache states and optimizing the representation for an efficient implementation of the function $acc : \mathbb{B}^n \rightarrow \mathbb{I}$, the application of regular partition and join operators also ensures a canonical representation of hardware states; because of the balancing operations, a particular hardware state always ends up in exactly one partition of an element of $\mathcal{D}$. This property allows for an efficient equality check of data flow elements by pairwise invocation of the equality operators of the two underlying domains on the contained partitions. It is most efficient if the number of partitions is small.

3.3 State traversal and performance

The state traversal for micro-architectural analysis on the domain $\mathcal{D}$ is implemented by repeated application of the function $step : \mathcal{H} \rightarrow \mathcal{H}$ to all elements of a domain element $D \in \mathcal{D}$. Partition and join functions are applied in each round of the traversal for balancing pipeline and cache states before applying the *step* function.

The proposed domain is most efficient if each cache state is associated with a large number of pipeline states. This allows for a small number of BDD operations which exploits the caching of intermediate results that is typical for BDD algorithms. Moreover, it significantly reduces the required number of cache updates since we perform a single cache update for all of the associated pipeline states. Note that a small number of partitions per domain element is also desirable.

A favorable relation between pipeline and cache states is maintained by the regular application of the join operator. The prior application of the partition operator minimizes the loss of cache precision and optimizes the BDD representation to allow for an efficient implementation of the function $acc : \mathbb{B}^n \rightarrow \mathbb{I}$. Its efficiency depends on the fact that

- the variables for addressing memory appear first in the BDD, and
- all encoded pipeline states access the same interval of addresses.

Hence, it suffices to enumerate the satisfying paths over the first m BDD variables. Let us consider the example depicted in Fig. 2. The example BDD shows only the first 6 state variables for accessing memory, i.e., we have $m = 6$. Note that in the full representation, the terminal node 1 would be replaced by a subgraph that represents the set of associated pipeline states. The BDD is evaluated by traversing the graph from the first variable node x_0 to one of the terminal nodes 1 or 0. Each variable node has two outgoing edges: the solid edge indicates that the variable has value 1, the dashed edge corresponds to the value 0. Nodes whose values do not influence the final result are omitted in the BDD (dont-care nodes). The terminal nodes represent the evaluation result. A path that ends at the terminal node 1 is called a satisfying path. It corresponds to one or several satisfying assignments of the variables. The satisfying paths over the example BDD of Fig. 2 are depicted in the first table of Fig. 3. To determine the interval that corresponds to a satisfying path, we set all dont-care nodes to 0 to obtain the lower bound (see table 3 in Fig. 3), and to 1 to obtain the upper bound (see table 2 in Fig. 3). Finally, we obtain the represented interval by taking the minimum and maximum of the intervals over all satisfying paths.

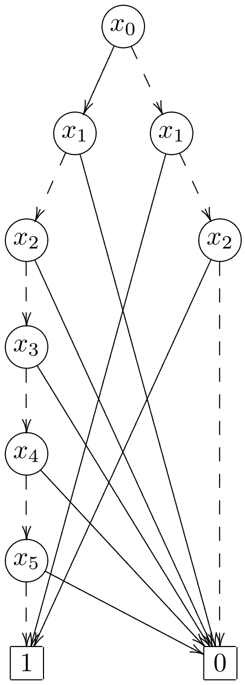


Figure 2 BDD representation of the memory access interval $[8, 32]$. In the full representation, the terminal node 1 is replaced by a subgraph that represents the set of associated pipeline states.

x_0	x_1	x_2	x_3	x_4	x_5
1	0	0	0	0	0
0	1	-	-	-	-
0	0	1	-	-	-

1. satisfying paths

x_0	x_1	x_2	x_3	x_4	x_5	ub
1	0	0	0	0	0	32
0	1	1	1	1	1	31
0	0	1	1	1	1	15

2. upper bound

x_0	x_1	x_2	x_3	x_4	x_5	lb
1	0	0	0	0	0	32
0	1	0	0	0	0	16
0	0	1	0	0	0	8

3. lower bound

Figure 3 Computing the lower and upper bounds of the intervals that correspond to the satisfying paths. The complete interval is then computed as $[\min\{32, 16, 8\}, \max\{32, 31, 15\}] = [8, 32]$.

The example shows that the interval can be computed from the BDD without enumerating all contained addresses. Note that the computational effort does not grow significantly if the interval shares a larger address prefix (using additional state variables $x_6, x_7, \ldots, x_m$ to address memory). The additional state variables either allow only for a single assignment, or most of them are dont-care nodes. The number of satisfying paths in the BDD will stay small.

4 Typical Cache Access Patterns

We experimented with 6 tasks of a commercial, safety-critical real-time software[1] to assess the correlation between memory accesses from different pipeline states. The tasks have been fully unrolled and annotated to avoid serious state explosion. The employed annotations specify ranges for register contents at selected program points to improve the precision of the value analysis and thereby reduce the reachable state space of the pipeline model. Note that full unrolling is not feasible for all software but required to obtain results with explicit-state implementations of very complex pipeline models. Otherwise, the analysis would not terminate in acceptable time because of state explosion. The following results have been obtained with the commercial, explicit-state pipeline model of the Motorola PowerPC

[1] Closed source and confidential.

755 [10, 1]. With full unrolling and annotations, all analyses terminate in less than 5 minutes running on an Intel i5 CPU at 2.67 GHz. We instrumented the pipeline model to print the following information for each access into a cached memory area:

1. Type of access, i.e., instruction or data.
2. Address and context of the currently analyzed basic block.
3. Cycle count since start of current basic block.
4. Accessed address or address range.

For each type of access, we collect all accesses with equal basic block address, analysis context, and cycle count. Symbolic pipeline analysis explores the model's state space cycle-wise in breadth-first order. Hence, all accesses in one set are issued from pipeline states in the same exploration layer. We partition the sets depending on the accessed addresses to obtain the number of different memory accesses from the same layer. The following tables list the results of this experiment. For each task (numbered $t_1, \ldots, t_6$) the first row gives the results for instruction cache accesses, whereas the second row reports the same information for data cache accesses.

Table 1 Avg. number of partitions per cycle.

t_1	t_2	t_3	t_4	t_5	t_6
2.19	1.51	1.94	2.03	2.13	1.52
1.38	1.11	1.29	1.35	1.35	1.02

Table 2 Max. number of partitions per cycle.

t_1	t_2	t_3	t_4	t_5	t_6
42	7	42	42	42	10
6	2	6	6	6	2

Table 3 Avg. number of states per partition.

t_1	t_2	t_3	t_4	t_5	t_6
17.24	35.35	25.49	25.61	35.63	19.96
10.82	28.07	20.76	25.94	25.61	7.87

Table 4 Max. number of states per partition.

t_1	t_2	t_3	t_4	t_5	t_6
4927	1311	8519	8190	8544	1091
1947	720	7140	7783	8115	268

According to Tab. 1, the average number of partitions is roughly 2. This number corresponds directly to the expected average number of partitions of an element of the proposed domain $\mathcal{D}$. Tab. 3 shows the average sharing, i.e., the number of pipeline states that can be encoded into a single BDD. The results indicate that the average relation between pipeline and cache states is roughly 18 : 1 (by dividing the average sharing of Tab. 3 by the average number of partitions of Tab. 1). The maximum number of partitions stays fairly small as shown by the results in Tab. 2. On the other hand, the maximum number of pipeline states per BDD can be quite large as shown by the results in Tab. 4.

All results indicate that the proposed domain operates on tuples with typical pipeline-cache relations between 1 : 1 and 8544 : 1, with an average of 18 : 1. These numbers hold under the assumption that the analysis maintains maximum cache precision. The proposed domain allows higher numbers of pipeline states per partition if caches are joined more aggressively. Larger numbers of pipeline states per partition can also be expected when the analysis encounters cases of imprecise information, e.g., about memory accesses.

5 Conclusion

We presented a new domain that integrates a symbolic exploration of abstract pipeline states with an abstract interpretation based domain for computing invariants of the cache state. Sets of pipeline states are stored in BDDs and manipulated symbolically using BDD operations. Abstract cache states are associated with sets of symbolically encoded pipeline states. Partition and join steps balance the representation to preserve a high analysis precision while avoiding an explicit enumeration of pipeline and cache states. Statistical data indicates that it is possible to maintain a favorable relation between pipeline and cache states, which allows us to reap the benefits of symbolic state traversal for pipeline analysis.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. We thank Daniel Kästner and Reinhold Heckmann for proof-reading this paper and Marc Schlickling for providing information about the Motorola PowerPC 755 pipeline model.

References

1 AbsInt. aiT WCET Analyzers. http://www.absint.com/ait, 2000.

2 R.E. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.

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

4 Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

5 C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

6 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

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

8 R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification, 1995.

9 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Saarland University, 2008.

10 Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

11 Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI*, pages 3–22. Springer Verlag, 2010.

12 Stephan Wilhelm and Björn Wachter. Towards symbolic state traversal for efficient WCET analysis of abstract pipeline and cache models. In *Proceedings of Seventh International Workshop on Worst-Case Execution Time Analysis*, July 2007.

13 Stephan Wilhelm and Björn Wachter. Symbolic state traversal for WCET analysis. In *International Conference on Embedded Software*, pages 137–146, October 2009.

Realism in Statistical Analysis of Worst Case Execution Times

D. Griffin and A. Burns

Department of Computer Science
University of York
York
YO10 5DD, UK
djg@cs.york.ac.uk burns@cs.york.ac.uk

Abstract

This paper considers the use of Extreme Value Theory (EVT) to model worst-case execution times. In particular it considers the sacrifice that statistical methods make in the realism of their models in order to provide generality and precision, and if the sacrifice of realism can impact the safety of the model. The Gumbel distribution is assessed in terms of its assumption of continuous behaviour and its need for independent and identically distributed data. To ensure that predictions made by EVT estimations are safe, additional restrictions on their use are proposed and justified.

Keywords and phrases WCET, Extreme value statistics, Gumbel distribution

Digital Object Identifier 10.4230/OASIcs.WCET.2010.44

1 Introduction

By their nature, hard real time systems have deadlines to meet, with consequences for failure. This leads to the problem of worst case execution time: finding out the minimum bound on a program's runtime. Unfortunately, in general this is an unanswerable question, as solving this would solve the halting problem. If restrictions are enforced on the program, then the question may be answerable - however, it may not be tractable.

The nature of these restrictions can be understood by noting that WCET estimation involves an element of model building, and therefore the work of Levins [10] is applicable. Levins describes a three-fold trade-off on useful models between *generality*, *realism* and *precision*, and argues that no useful model can maximize all three of these properties. Whilst Levins did not define these terms, assuming the meanings to be obvious, they have come to mean the following:

- *Generality*: The degree to which the model is applicable to multiple situations in the real world.
- *Realism*: The degree to which the model accurately represents the actual phenomena occurring in the real world.
- *Precision*: The degree to which error bounds are minimised when comparing the model's predictions with the real world.

The term "useful model" was later defined as one which is understandable, measurable, and computationally soluble [12]. Later work by Bullock and Silverman [3] named these three properties the models *tractability*, and constructed an argument for a four-fold tradeoff between tractability and the original properties of a useful model defined by Levins. Within this four-fold tradeoff, Levins useful models are situated at the threshold between intractable and tractable models.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 44–53
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Levins' philosophy is demonstrated in current approaches to WCET estimation such as abstract interpretation [7] which sacrifices precision to gain an approach which is general and realistic - and the lack of precision manifests itself in a relatively high overestimation of the true WCET. Unfortunately, the requirement to guarantee the safety of the estimation, that the WCET is bounded from above, not below, requires an element of precision. Further, current research into abstract interpretation is increasing the precision of the method, which according to Levins will require another aspect to be sacrificed. One present approach achieves this by sacrificing some realism by using a heuristic [14]. Other approaches may sacrifice generality, or may simply build a more precise and more complicated model which sacrifices tractability.

Statistical analysis, as proposed by Edgar and Burns [5, 6] is a method which takes a radically different approach to abstract interpretation. Statistical analysis sacrifices realism for generality and precision, by using extreme value theory (EVT) statistics [9]. EVT is a collection of statistical models which are suited to accurately predicting the tail end of a distribution. By fitting one such model, the Gumbel distribution, to observed data the probability of failing to meet a given execution time can be found, and this can be used as the foundation of a probabilistic real time system [2]. Clearly, EVT is not a realistic model of a computer system, but it can model observed behavior precisely. Statistical analysis was studied further by Hansen, Hissam and Moreno [8], who modified the work to fit more accurately with the original design of EVT. The same modifications also produce results in the form of probability of failure in a given time period, as preferred by the testing community [4].

This paper aims to look at an issue which has not yet been adequately addressed: the impact of the lack of realism in the model on the safety of Gumbel-derived estimates. Primarily, this manifests itself in subtle differences between the Gumbel distribution and the properties of an actual system. Sections 2 and 3 detail two significant lapses of reality in the model, and demonstrate how unsafe predictions could be made using statistical analysis. Section 4 details methods to take into account these lapses of reality, either by proving that they do not apply or adapting the application of statistical analysis to compensate. Conclusions are presented in Section 5.

2 Continuous vs. Discrete Distributions

The Gumbel distribution, along with any other continuous probability distribution, makes an assumption that all values are possible. This is clearly unrealistic; looking at the control flow graph of a program will show that a program can not terminate at any point. Instead, programs perform computation for a period of time, and then may stop or perform more computation. For instance, consider the program in Figure 1, being run on a simple processor with no features to speed up execution. The function F refines a result until either the result is accurate enough or the routine hits a recursion depth limit. This program may or may not exhibit exponential decay in the probability of its execution times depending on the nature of G, the refinement function, and the input x. However, it has the property that it cannot terminate during an execution of G, or the accuracy test. This means that the only times it can terminate are the times expressible as:

$$\sum_{i=0}^{j} rt(f(x_i)) \text{ where } j \in [0,5], \text{ and } x_i \in Dom(F)$$

Assuming that the domains of the functions F and G are finite, then the program may

```
$F(x, d)$
$x = G(x)$
if $x$ has sufficient accuracy or $d \geq 5$
    return $x$
else
    return $F(x, d + 1)$
```

Figure 1 A program which does not have a continuous distribution of runtimes

only terminate at finitely many times. Further, the set of all possible termination times need not occupy the entire interval of termination times. For instance, F and G may be written so the program takes a multiple of 5 cycles to complete. Then the program will not terminate on any cycle which isn't a multiple of 5.

This becomes a problem because the Gumbel distribution assumes that the program can stop at any point, and hence will produce results which do not make sense. Using the previous example, it is known that the probability of termination by cycle 9 is the same as termination by cycle 6 - if the program hasn't terminated before cycle 6, it cannot terminate until cycle 10. The prediction of the Gumbel distribution differs, as shown in Figure 2. In fact, the Gumbel distribution produces values lower than the actual distribution - and these values are unsafe.

Hence the problem is as follows. By using a continuous function to model the execution time of the program, an assumption is made on the program that it can terminate at any point. Whilst any synchronous processor has some form discrete time, therefore invalidating this assumption, such small errors would pass unnoticed. The major issue is that due to how programs are written, this error could be a noticeable amount of time. For example, suppose that the function G took a multiple of a million cycles to run. As embedded processors operate in the MHz frequency range, this could amount to delays of a noticeable fraction of a second.

3 EVT assumes the IID property

EVT also makes the assumption that the data to be modeled is i.i.d., or independent and identically distributed [9]. As an example, consider the original purpose of the Gumbel distribution: modeling floodwater levels - a safety critical situation, where reliable results are needed. Floodwater data is normally independent; provided the landscape is unchanged and flood defences repaired but not enhanced, one flood does not affect another. If this is not the case, then future floods are said to depend on the flood which caused this change. Similarly, these future floods will also have a different distribution to the previous floods, due to the changes effecting how much water is needed to cause flooding. Hence, these future floods do not follow the same distribution as the past floods, and so are not identically distributed.

In statistics, the i.i.d. assumption simplifies things immensely, if it can be made. Unfortunately, the execution times of programs need not be independent or identically distributed.

An obvious example of runtimes being dependent is that of amortised data structures. In this case the runtime of adding an element to the data structure depends on the current state of the data structure. Avoiding amortised data structures is not enough though; depending

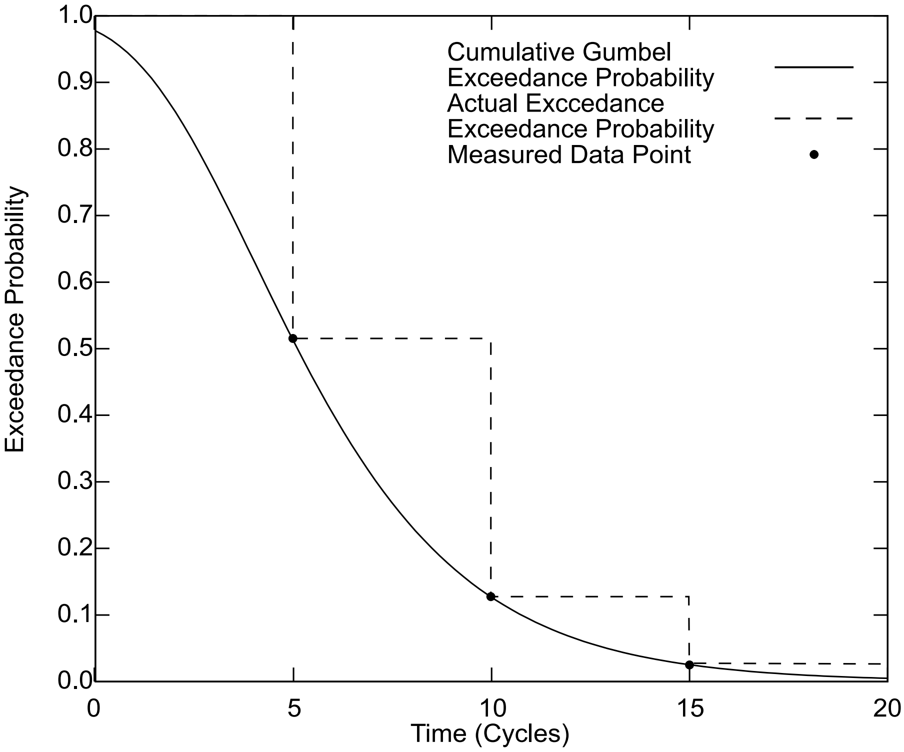


Figure 2 An illustration of how the Gumbel distribution can infer unsafe values due to the implicit assumption that a program may stop at any time during execution.

on testing methodology, almost any program can have dependent runtimes, simply through an on-chip cache. If the program can leave the cache in multiple states depending on its input, then each runtime is dependent on what was run before it - and such a dependence may not be trivial to model. For instance, a long runtime may mean that more parts of the instruction code have been loaded into cache, decreasing the next runtime. Alternatively, it may mean that this run of the program performed a lot of cache thrashing - and hence leaving the cache in a less useful state for the next run of the program.

Further, some systems may necessarily have dependent runtimes, especially in the case of systems which manipulate a value in the real world. As an example, take an aircraft flight control system. As an input, the control system will consider the present velocity of the aircraft and produce an output which is a modification to that velocity. Hence the input, and therefore runtime, of any run of the system will depend on the outputs of every previous run.

A similar argument can be used to reason that program runtimes need not be identically distributed, in that each run of the program has the potential to "change the world". However, it turns out that the problem of runtimes not being identically distributed is more widespread. Suppose $X(n)$ is the uniform random distribution over the closed interval $[1, n]$, and that programs A, B produce different runtime distributions when run with data sampled from $X(100)$. Then define program C as follows:

$$C(n) = \begin{cases} A(\lfloor n/2 \rfloor) & \text{if } n \text{ is even} \\ B(\lfloor n/2 \rfloor) & \text{if } n \text{ is odd} \end{cases}$$

Then if C has its runtimes sampled with data from $X(200)$, the runtimes will follow two distributions - one for those which have odd input data, and one for those with even input data. Hence the runtimes cannot be treated as being identically distributed at face value. It is possible to combine multiple distributions into one, but this can cause a loss of accuracy when fitting the Gumbel distribution. Further, it would be necessary to ensure that all possible distributions of runtimes were found. Given that the number of paths through a program is potentially large, and each path potentially leads to a different distribution, this would cause an unacceptable amount of testing to be necessary.

It should also be noted that this observation correlates with the results presented in Edgar's work [5]. For experiments on branch prediction there is a strong argument for the results being i.i.d.: that the effects of other unknown processes on the branch prediction experiment can be modeled by some i.i.d. random variable. In turn, this leads to highly accurate results. In more complicated experiments, such as bubble sort, there is no such argument as the amount of work the algorithm does is a non-trivial property of the input. Consequently the results of Gumbel prediction are much poorer in the bubble sort experiment [5].

In related work, Petters [13] also stated that the i.i.d. assumption needs to hold for EVT statistics to be valid. Petters' approach is to use statistical methods to gain a confidence value on predicted runtime of "measurement blocks" of code, and combine these to get an overall WCET. To ensure the i.i.d. assumption holds, Petters proposes to randomise any potentially unknown element of the processor state at the beginning of each measurement block. By using these measurement blocks, the problems outlined above are avoided as there are limited paths through the code, and within these blocks hazards are encountered with fixed probabilities determined by the initial randomisation. The potential downside to Petters' work is that it is assumed that the measurement blocks are independent. Whilst for Petters' work, on modelling processor features, it can be ensured that measurement blocks are independent, it may not be a practical assumption when modelling entire systems.

4 Additional Properties Needed to use EVT

Given the issues raised in the previous sections, it is apparent that to make use of EVT it is necessary to take measures so that it is known that the lack of realism in the model does not produce unsafe estimates.

There are two potential solutions available. The first is to prove that the problems either do not apply, or are bounded. The second is to adapt the use of statistical analysis so that it does not encounter any problems.

4.1 Proving Strategies and Bounding Undesirable Behaviour

One strategy to avoid problematic effects is to prove that they do not exist, or to prove a bound on them. Neither of these strategies can be automated, as mathematical proof is not automatable. Fortunately, there are some general techniques which can be used, which are outlined here. The techniques focus on producing some form of bound on undesirable properties.

To cope with the effects of approximating a discrete distribution with a continuous distribution, there is a simple method. This is to determine the maximum error this introduces and to add this error to any prediction. This seems as if it could be difficult to accomplish, as the maximum error is not known; however, there is a simple method available which does not require this information. Instead of modeling the observed data directly, first convert the data to exceedance probabilities, such that value x_n is exceeded with probability p_n, for $n \in [1, k]$ and the x_n being original data points. Then model the data series of value x_n occurring with probability p_{n-1}, with $p_0 = 1$. This is illustrated in Figure 3 and moves the Gumbel estimate above the safety line by the minimum amount required to guarantee safety. The downside to moving the points in this way is that the estimates are pessimistic almost everywhere.

Another method to deal with errors from approximating a discrete distribution with a continuous distribution would be to try and argue that the effects are negligible. For instance, in the case when only a single path is being considered the error can be bounded by the largest single processor delay. This delay is likely to be measured in tens of processor cycles, and in if the processor speed is measured in MHz then the errors involved are likely to be negligible. Similar arguments could also be made if the different paths through the program have similar lengths and experience similar amounts of randomly-modeled hazards.

Proving the i.i.d. assumption is hard to do conclusively, with caches being an example of a common processor feature which causes the i.i.d. assumption to be violated. However, in some systems, particularly soft real time systems, it may be adequate to perform statistical analysis of the test data to determine if the i.i.d. assumption appears to hold. A statistical test for independence is as follows: Assume the null hypothesis that the execution times are independent of the system's previous actions. As the system's previous actions are determined by the system's input, it is sufficient to look for correlation between the runtime of a test and the inputs to previous tests. If correlation exists then the null hypothesis can be disproved; if correlation does not exist it only indicates that the test data does not disprove the null hypothesis, and that over the entire set of test data the data appears independent to some degree of confidence. The test can be improved by splitting it into subtests of contiguous test data; this would prevent small runs of of dependent results from being hidden by a large quantity of independent results.

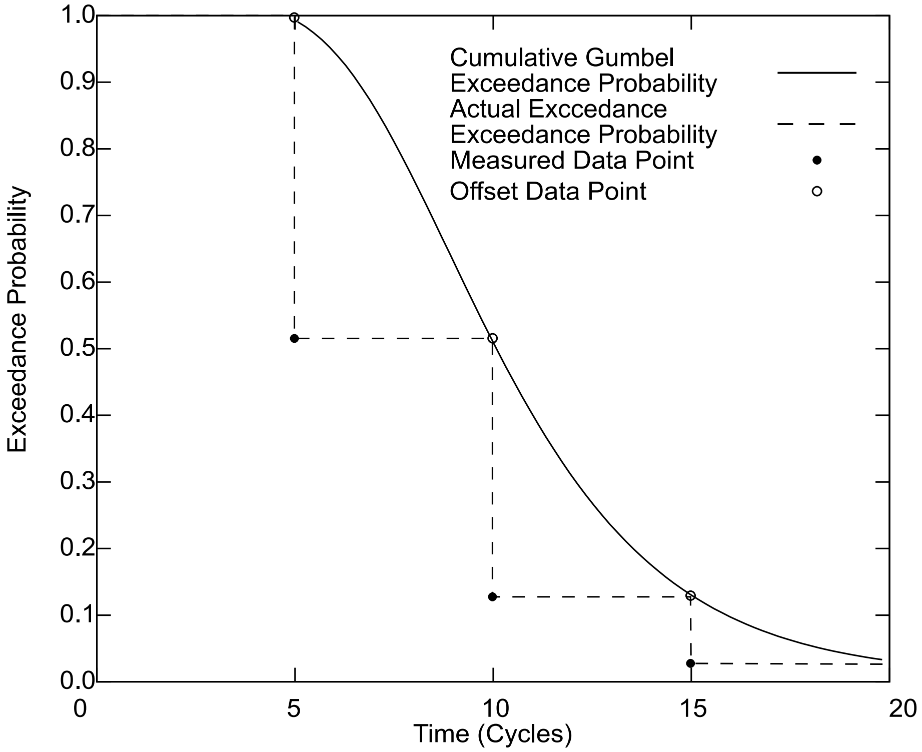


Figure 3 An illustration of how to offset the Gumbel distribution to guarantee safe values

Another tactic to take into account dependence would be to bound it. This could easily be accomplished by a periodic reset of the system, which would break dependencies. Then the drift in execution times over this period could be found through a number of methods, including statistical analysis, as these are i.i.d. and very close to being continuous. For some systems, for instance the previously mentioned aircraft control system, a complete reset of the system may be impossible. In this case an appropriate strategy could be to periodically reset those resources which can be reset, and perform statistical analysis to gain some confidence that the remaining inputs to the system do not cause any dependencies between job executions.

Proving the identically distributed assumption is harder. The problem is that for every path through the program there will be a different set of hazards, leading to a potentially different distribution, and that it is difficult to argue that the worst case distribution is represented in the test data. If the worst case distribution is not represented, then statistical analysis cannot take it into account when combining all observed distributions into a Gumbel distribution. The simplest method here would be to ensure some level of code coverage, such as modified condition/decision coverage [1], and to ensure that each different path identified by the code coverage technique has sufficient samples taken from it to form a distribution.

4.2 Adapting Statistical Analysis

If it is not possible to bound the effects of not meeting the i.i.d. assumption, then the application of statistical analysis must be adapted instead. For this, it is necessary to look at what effects are being modeled, and identify what could cause problems. Given a sequential program and a system to run it on, it is reasonable to say that the execution time of the program depends on three properties:

- The input to the program.
- The initial state of any resources the program uses.
- Competition for resources from other programs (in preemptive or multiprocessor environments).

The third of these can be ignored, provided that testing is carried out in a representative environment including the programs which compete for the resource. However, it is necessary to enforce that programs do not directly communicate with each other, as this can introduce non-i.i.d. behavior. In such a case, it would be possible to use statistical analysis to predict the WCET of a group programs which do communicate heavily with each other, as then the dependencies become internal and invisible to the model.

Similarly, there is a simple solution to the second of these properties: perform a reset of the processor and shared resources between tests. Unfortunately, such an assumption may not be realistic or desirable in the real world, so an alternative would be to test multiple subsystems which all share the same resources at once. If such tests were randomly interleaved, then it becomes valid to treat the initial state of the resources as another i.i.d. random variable, because for each test of a subsystem, any other test on any subsystem could have been carried out before, which effectively randomises the shared resources.

This only leaves the fact that input can effect runtime, leading to non-identically distributed runtimes, as a problem. The most obvious solution to this would be to fix the input such that it forces the worst case path through the program. The fixed input would guarantee that the measurements are sampled from the same distribution. The major concern with this approach is that due to low level processor details, such as floating point division, it is conceivable that using a single input will not be able to find the WCET. Hence it makes sense to widen the values used as input, when sampling, to the set of all inputs which lead to the worst case path through the program.

The biggest issue with the above remedy is that it requires knowledge of the worst case path through the program, but the worst case path through the program is not known - or WCET analysis wouldn't be necessary. However, other analysis methods may be able to determine a set of candidate paths through the program, of which one is the worst case path. This can be seen in abstract interpretation [7], which uses abstract states consisting of many concrete states to determine the worst case path. Statistical analysis could be used in this case to analyse each path in such a set individually to determine which path is the worst case, essentially reversing the abstraction. It may be that statistical analysis would indicate that all of the paths in the worst case abstract state present a better case than those in some other set, in which case the paths to be explored by abstract interpretation would have to be extended to this other set.

An alternative method of making the identically distributed assumption hold would be to use external measurements, such as the number of instructions executed and the number of cache misses, to create a catergorisation scheme. Provided that the chosen measurements lead to categories whose members are sampled from the same or similar distributions, the identically distributed assumption should hold. Then it becomes possible to work out a WCET time by applying statistical analysis to each category and picking the maximum. Alternatively, the categories could form the basis for using parametric WCET analysis techniques [11] to further enhance the precision of the results.

If the i.i.d. assumption is satisfied by using some of the methods outlined above, then it only remains to determine that the errors associated with fitting a smooth curve to the distribution are small and that the safety of the model is preserved. Fortunately, this turns out to be relatively straightforward: given the restrictions to make the measurements meet the i.i.d. requirement, the measurements should be similar enough that a significant error, of the type seen in Figure 2, cannot exist.

5 Conclusion

The use of statistical estimation for WCET is a powerful method, and can be used to model WCET with great accuracy [5, 6, 8]. However, as statistical estimation sacrifices realism it is necessary to take measures so that the safety of the model is not an unintended casualty of this sacrifice. To ensure the safety of the model it is either necessary to prove that the assumptions of EVT statistics hold or to adapt the application of statistical analysis. Adaption requires an additional set of restrictions on the measurements used to generate the statistical model, but in turn these should produce a more sound model.

References

1 T. Badgett, T. M. Thomas, and C. Sandler. *The Art of Software Testing.* John Wiley and Sons, Inc, second edition, 2004.

2 G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288, 2002.

3 S. Bullock and E. Silverman. Levins and the legitimacy of artificial worlds. In N. David, editor, *Third Workshop on Epistemological Perspectives on Simulation*, 2008.

4 R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Softw. Eng.*, 19(1):3–12, 1993.

5 S. Edgar. *Estimation of Worst-Case Execution Time Using Statistical Analysis.* PhD thesis, University of York, 2002.

6 S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings IEEE RTSS 2001*, 2001.

7 C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

8 J. Hansen, S. A. Hissam, and G. A. Moreno. Statistical-based WCET estimation and validation. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, Dublin, Ireland*, volume 252. Austrian Computer Society, 2009.

9 S. Kotz and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* Imperial College Press, 2000.

10 R. Levins. The strategy of model building in population biology. *American Scientist*, 54(4):421–431, 1966.

11 B. Lisper. Fully automatic, parametric worst-case execution time analysis. In J. Gustafsson, editor, *WCET*, volume MDH-MRTC-116/2003-1-SE, pages 99–102. Department of Computer Science and Engineering, Mälardalen University, 2003.

12 J. Odenbaugh. The strategy of "the strategy of model building in population biology". *Biology and Philosophy*, 5:607–621, Nov 2006.

13 S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Institute for Real-Time Computer Systems, Technische Universiät München, 2002.

14 R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212, New York, NY, USA, 2007. ACM.

Hybrid measurement-based WCET analysis at the source level using object-level traces

Adam Betts[1], Nicholas Merriam[1], and Guillem Bernat[1]

1 **Rapita Systems Ltd.**
IT Centre, York YO10 5DG, UK
{abetts,nmerriam,bernat}@rapitasystems.com

Abstract

Hybrid measurement-based approaches to worst-case execution time (WCET) analysis combine measured execution times of small program segments using static analysis of the larger software structure. In order to make the necessary measurements, instrumentation code is added to generate a timestamped trace from the running program. The intrusive presence of this instrumentation code incurs a timing penalty, widely referred to as the probe effect. However, recent years have seen the emergence of trace capability at the hardware level, effectively opening the door to probe-free analysis.

Relying on hardware support forces the WCET analysis to the object-code level, since that is all that is known by the hardware. A major disadvantage of this is that it is expensive for a typical software engineer to interpret the results, since most engineers are familiar with the source code but not the object code. Meaningful WCET analysis involves not just running a tool to obtain an overall WCET value but also understanding which sections of code consume most of the WCET in order that corrective actions, such as optimisation, can be applied if the WCET value is too large.

The main contribution of this paper is a mechanism by which hybrid WCET analysis can still be performed at the source level when the timestamped trace has been collected at the object level by state-of-the-art hardware. This allows existing, commercial tools, such as RapiTime, to operate without the need for intrusive instrumentation and thus without the probe effect.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.54

1 Introduction

The task schedule in a real-time system is responsible for allocating the CPU's time to a number of single-threaded tasks. All scheduling algorithms, and the schedulability tests that check their feasibility, assume that the **Worst-Case Execution Time** (WCET) of each task is available as a fixed value.

Deriving the *actual* WCET completely automatically with a tool is impossible in the general case, not least because such a tool would also solve the halting problem, see [12]. However, practical experience has shown that a useful *estimate* of the WCET can be derived using a tool with a little guidance from the user.

Most often, industry uses the WCET estimate to validate compliance of tasks with their execution-time budgets, assigned early in the development process to allow construction of the task schedule. This validation is done by recording end-to-end execution times under quite extensive and demanding testing conditions; the longest time is generally referred to as the **High-Water Mark Time** (HWMT). The underlying premise of this approach is that the testing regime is representative of real system operation and that, with enough testing, the HWMT lies in close proximity to the actual WCET. However, there are a number of disadvantages of such a simplistic approach:

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 54–63
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- No understanding is given regarding the path through the software which leads to the HWMT and engineers therefore lack insightful knowledge of where time is spent. This information is crucial in optimisation efforts, particularly when the HWMT exceeds the budget of a task.
- Although pessimism is eliminated (either the test harness captures the actual WCET or an uncomputable degree of underestimation exists), a bound on the actual WCET cannot be guaranteed without full path coverage and, perhaps more significantly, *full state coverage* at the hardware level. Acknowledging this issue, industry adds some percentage to the HWMT in order to bypass any underestimation, and considers this as the WCET estimate. Even with this additional step there is still no guarantee that the actual WCET is bounded; it could even lead to underutilisation of system resources if, in fact, the actual WCET has been captured but the safety margin is too excessive.
- In safety-critical systems, MC/DC [5] is normally the coverage metric that needs to be satisfied before testing halts. In effect, properties affecting the execution time, such as hardware effects and loop iterations are neglected. This is a considerable anomaly given the increasing complexity of modern embedded hardware and the jitter it introduces into the execution time profile of a task.

An alternative to high-water marking, offered by **Hybrid Measurement-Based Analysis** (HMBA), is to measure execution times of small program segments and then combine these data in a final calculation stage, to deliver a WCET estimate. The necessary measurements are extracted from a timestamped trace of execution, which a running program emits when *instrumented*. Because instrumentation code is traditionally compiled into the executable, it has been *intrusive* by nature. On the one hand, this provides portability and flexibility. On the other hand, it manifests the **probe effect** where normal (i.e. without instrumentation) register and cache usages are displaced, a timing penalty is incurred, and overall code size increases.

Recent years, however, have seen an increasing number of embedded micro-controllers feature some level of trace capability (Nexus [1] adopted by Freescsale, ARM Embedded Trace Macrocell [8], and others) without the requirement for software instrumentation. A suitable debugger can then reconstruct exactly the sequence of instructions executed together with timestamps. This trace of instructions and timestamps provides an ideal foundation for WCET analysis at the object-code level, having great detail while eliminating the probe effect.

But object-level WCET analysis has two major drawbacks. First, many current HMBA techniques [6, 7, 3], and commercial HMBA tools such as RapiTime, derive their program model from hierarchical source code structures. They cannot, therefore, directly analyse the hardware-generated trace of object code. Second, even if that first obstacle were overcome, it remains expensive for engineers to interpret the results, such as which code is on the worst-case path, since their expertise concerning the application is usually at the source code level.

In this paper, we present a mechanism by which HMBA can still be performed at the source level when the timestamped trace has been collected at the object level by state-of-the-art hardware. This allows existing HMBA techniques to operate without the need for intrusive instrumentation and thus without the probe effect.

The remainder of this paper is structured as follows. Section 2 gives more background on HMBA, in particular discussing mechanisms available to obtain timestamped traces. Section 3 then shows how an object-code trace can be utilised at the source level and discusses future issues to be resolved. Next, Section 4 demonstrates the improvement gained by the proposed method over existing intrusive instrumentation. We finally conclude in Section 5.

2 Background and Related Work

2.1 Instrumentation and Tracing

As noted above, HMBA relies on **instrumentation points** (ipoints) to collect measurements. Upon execution during testing, these ipoints emit (integer) identifiers and are time stamped accordingly, resulting in a timestamped trace of execution. The trace is subsequently parsed to extract both the WCETs of program segments and other path-related information such as loop bounds (if required).

There are a number of mechanisms available to generate and extract traces:

- Simulation: Cycle-accurate simulators, such as SimpleScalar [2], allow individual instructions to be traced (through the program counter) whereby the simulator provides the time stamp. However, because a simulator is a hardware model, this method encounters problems associated with producing a cycle-accurate processor model. This is often very difficult due to missing or incorrect information [9].
- Software: A tailored ipoint routine is inserted into the source code at particular locations. When an ipoint is hit during execution, it is time stamped on target; traces are stored internally in a memory buffer to be downloaded on test completion. The advantage of this approach is that porting to new architectures is relatively straightforward. However, this results into the probe effect.
- Software/Hardware: This is similar to software only instrumentation, except that the ipoint routine writes its identifier to an I/O port on target. The port is monitored by an external capture device, e.g. a logic analyser, which timestamps ipoints off target as they appear and also serves to store the timing traces. Penalties associated with the probe effect are minimised because the ipoint routine can be reduced to a few instructions. However, the target must have available and accessible pins to emit the data, which is not always practical with more advanced processors. It may also be necessary to disable the cache so that the data written by the ipoint routine is observed on the bus.
- Hardware: The probe effect can be prevented with the assistance of on-chip debug interfaces. In these cases, the trace data are either written to an on-chip trace buffer for subsequent download, or exported directly in real time through an external port. In order to limit the size of traces, only the program flow discontinuities are monitored, i.e. taken jumps. However, bandwidth remains the major technical obstacle because the port or debugger must keep pace with the rate at which trace data are produced; otherwise, blackouts arise in which parts of a timing trace are overwritten and essentially lost.

There is clearly a trade-off in using any of the above trace generation methods. On the one hand, source-level instrumentation provides greater flexibility and is often the most convenient, but it is handicapped by the probe effect. On the other hand, less intrusive instrumentation normally demands some type of hardware support.

2.2 Hybrid Measurement-Based Analysis

Although end-to-end testing and HMBA are closely linked, in that they both utilise test vectors to stress execution times, the latter has the following principal advantages:

- Piecing together measured execution times of small program segments with path-specific information in the WCET calculation alleviates the test harness in two ways. First, it eliminates the requirement to find a test vector that causes all loops to iterate through their maximum bound *simultaneously*. By using a program model, HMBA instead allows loop bounds obtained from SA or end users to be included *a posteriori* to the measurement stage. Second, the test harness need not attempt to trigger the WCET of program segments in the *same* run, as HMBA multiplexes this information from all executions before carrying out the WCET computation. Obviously, coverage remains an issue since the units of computation must be stressed adequately enough to represent worst-case behaviour, although how this is realised is an open research question.

- Increasing the level of instrumentation beyond the start and end of the program allows valuable information to be extracted, including: the functions executed most frequently on the path leading to both the HWMT and the computed WCET estimate; the number of observed loop iterations; which sections of code have been covered by the test harness. As mentioned above, this information is essential to engineers as it provides insights into the non-functional properties of the code.

Once the timestamped trace has been generated, the calculation engine is tasked with producing a WCET estimate; how this is done depends on the type of program model. Much of the research on calculation engines has assumed a graph-based program model, such as the **Control Flow Graph** (CFG), chiefly because CFGs can easily model arbitrary control flow, including `goto`-littered code; comparatively, the **Abstract Syntax Tree** (AST), which is the *de facto* tree-based program model, fails or struggles to do so as the hierarchical relation between program segments is absent.

However, tree-based approaches based on the AST retain a number of advantages. First, in comparison to the linear programming [13] and path-based [14] calculations of graph-based program models, low computational complexity is incurred. Efficiency of the calculation is a specific concern when WCET analysis is integrated into an interactive environment [10] or when the system has a large code base. Second, in addition to integral values, execution time profiles derived from measurements can be combined to produce probabilistic WCET estimates [3, 4]. Third, it is straightforward to relay the longest path returned by the calculation engine onto the source code.

3 Object-Code Tracing

The normal work-flow of a source-level HMBA tool is to insert ipoints into the code while the program model, i.e. the AST, is being constructed. As ipoints are positioned, they are also assigned identifiers. When executed, these ipoints produce a trace of the form `(identifier, timestamp)`. However, the hardware tracing mechanisms mentioned in the previous section instead provide a trace of the form `(address, timestamp)`, where `address` is the destination of a branch; that is, the identifier of an ipoint is an address. Thus a source-level HMBA tool is usually unable to consume a trace in this format because there is no correspondence between the ipoint identifiers assigned and the addresses observed.

Thus the first issue to overcome is the ability for each inserted ipoint to use an address as its identifier. This can be done using global assembly labels[1]. That is, global assembly labels are generated for each inserted ipoint using a macro, an example of which appears in Figure 1. The resulting addresses of the labels are then passed back to the HMBA tool so that it can replace the original identifiers of ipoints. With this small step, the HMBA tool is

[1] Such a feature has already been implemented by RapiTime in the MERASA project — see *http://www.merasa.org*

```
#define RPT_Ipoint( I )                                 \
({ asm volatile (".globl __rpt_ipoint_" # I "\n"        \
                 "__rpt_ipoint_" # I ":" );             \
 })
```

Figure 1 C Macro to Generate Assembly Label Ipoint

able to parse a trace in (address, timestamp) format and therefore perform the WCET analysis.

It would appear that debug information generated by the compiler could instead be used to map traces to the source level, rather than global assembly labels. However, in industry, debug builds are typically only available very early in the development process; this solution is independent of debug information being available.

The second issue is that hardware debug interfaces only record a sequence of branch destinations, and not every address which corresponds to an ipoint label. The solution is to analyse the disassembly for branch instructions and then interpolate the missing instructions from the trace, recording only those that correspond to ipoint addresses. With both of these steps, therefore, a HMBA tool is able to consume an object-level trace as produced by a hardware debug interface.

3.1 Discussion

There are several outstanding issues with the proposed mechanism as summarised in the following discussion.

3.1.0.1 Processor limitations:

Evidently, our technique is limited to processors that provide tracing facilities. This implicitly excludes high-end PowerPC architectures which are widely used in the avionics industry and where there is a strong need for WCET analysis for certification reasons.

3.1.0.2 Loop unrolling:

Optimising compilers often unroll loops [11], in effect replicating the body of a loop a particular number of times and adjusting the control logic as necessary. This would generate multiple copies of a global label, which is illegal assembly, and consequently such optimisations have to be disabled. In any case, such optimisations are normally disabled in safety-critical systems.

3.1.0.3 Ipoint locations:

Knowledge of the object code also assists in improving the accuracy of source-level instrumentation. For instance, consider the source code in Figure 2. Here the intention is to observe the start execution time of `check_sum` with the ipoint `2147483643` (through the function `RPT_Ipoint`). However, in the compiled object code, shown in Figure 3, this ipoint appears at address `003f98b4`, which is six instructions after the start of `check_sum`. This can be resolved by instead assigning the start address of the basic block (which contains the ipoint label) as the ipoint identifier, rather than the raw address of the compiled label. In this example, therefore, the ipoint should be assigned identifier `003f989c`.

```
Uint8 check_sum( Uint8 * msg, Uint8 len )
{
  RPT_Ipoint( 2147483643) ;
  Uint8 c, i;
  c = 0xAA;
  ...
```

Figure 2 Example Instrumented C Source Code

```
003f989c <check_sum>:
  3f989c:       94 21 ff d0     stwu    r1,-48(r1)
  3f98a0:       93 e1 00 2c     stw     r31,44(r1)
  3f98a4:       7c 3f 0b 78     mr      r31,r1
  3f98a8:       90 7f 00 18     stw     r3,24(r31)
  3f98ac:       7c 80 23 78     mr      r0,r4
  3f98b0:       98 1f 00 1c     stb     r0,28(r31)

003f98b4 <__rpt_ipoint_2147483643>:
  3f98b4:       38 00 ff aa     li      r0,-86
  3f98b8:       98 1f 00 09     stb     r0,9(r31)
  3f98bc:       38 00 00 00     li      r0,0
  3f98c0:       98 1f 00 08     stb     r0,8(r31)
  ...
```

Figure 3 Example Instrumented Disassembly

Note that, when the intention of the ipoint is to record the time at which flow returns to the calling function, it is more appropriate to use the last address of the containing basic block as the identifier of an ipoint.

The next step in this work is to distinguish between ipoints corresponding to the beginning and end of function execution and all others, assigning each of the former its optimal identifier.

3.1.0.4 Trace size:

Tracing at the object-level typically results in very rapid generation of data. For instance, a processor running at 200MHz, with an average of one branch every 10 instructions and using 64 bits to record each branch timestamp, would generate 160MB in one second. A one-hour test would generate around half a terabyte of data. Despite the fact that storage costs continue to fall, object-level trace data remains difficult and expensive to archive and analyse. A key step forwards is to interpret the data in real time, removing the need to store all the trace data.

In 2010, Rapita Systems Ltd is starting a development project, assisted by the UK regional development agency Yorkshire Forward, to bring together the latest hardware and software techniques to allow real-time analysis of trace data.

4 Evaluation

To show the benefit of the proposed approach, the techniques described in the previous section were implemented in the RapiTime toolchain.

We analysed a synthetic in-house benchmark of 226 lines of C code, comprising six functions, whose functionality is to receive and respond to a number of different kinds of messages. The benchmark was instrumented using the C instrumenter of the RapiTime toolkit, which inserted a total of 28 ipoints. Compilation of the instrumented source was done through a GCC cross-compiler for the PowerPC architecture.

The binary was then wrapped in a test harness and executed on a Freescale MPC565 with a Wuerz evalutation board. The timestamped trace was obtained through a Lauterbach PowerTrace debugger via a Nexus debug port. Two such traces were obtained: one using an intrusive tracing mechanism and the other using the approach presented in this paper.

Each of the traces were parsed by RapiTime to produce a WCET estimate and other timing information for each function. A screenshot of the RapiTime report viewer showing results of the analysis appears in Figure 4. The most important information appears in the two stacked views in the centre:

Top view: This shows various execution time data, such as the longest execution time and the WCET estimate. The root function under analysis is `message_receive`: the highlighted times show its longest end-to-end execution time.

Bottom view: This displays the source code and the relative locations of ipoints. In Figure 4, there is an ipoint at the start of `message_receive` with identifier `4168328`, corresponding to the address `3f9a88`.

A more thorough presentation of the results appears in Table 1, which shows the WCET estimate for each function in the application obtained from object-level and source-level instrumentation. As can clearly be observed, every function benefits from the probe-free instrumentation. For the root function `message_receive`, the WCET is reduced from 20.339 microseconds to 11.863 microseconds, the former owing to instrumentation overheads.

This amounts to a $\approx$ 42% decrease in the overall WCET estimate when using object-code instrumentation, which is a substantial benefit for a relatively small application, thus underlining the benefit of adopting hardware debug interfaces in HMBA.

Function	WCET Object Instrumentation	WCET Source Instrumentation
`message_receive`	11.863	20.339
`check_sum`	7.232	8.732
`count_set_bits`	4.940	9.232
`process_in_buffer`	6.858	11.214
`send_message`	0.746	0.804
`simulate_save_to_flash`	0.457	0.839

Table 1 WCET Estimates (in Microseconds) for Each Function in Analysed Program

It should be noted, however, that the overheads of the intrusive instrumentation are unusually high in this application. This is because the code base itself is fairly trivial, containing several tight loops. Future work will assess the impact of the proposed technique on a range of more realistic and, in particular, larger examples.

5 Conclusions

State-of-the art HMBA tools such as RapiTime automatically insert software instrumentation points into the source code while constructing the program model on which they base the WCET calculation. The biggest drawback of this approach is that the instrumentation adds an overhead. The probe effect is a key weakness of HMBA tools and an objection to their uptake, with the overheads not only disturbing the accuracy of the results but even making the software too slow and/or large to run in the target test environment.

This can be circumvented by instead utilising the trace facilities provided by hardware debug interfaces. Using object-level tracing, however forces the analysis to that level. This is especially inconvenient for commercial tools, which normally present the calculated data at the source level, since this is where most engineers retain expertise. This paper presented a solution to the problem by showing how to match the addresses of instrumentation points in the program disassembly to source lines.

We showed some initial results using the presented approach: for a small application, there was a $\approx$ 42% reduction in the WCET estimate in comparison to using intrusive instrumentation. The ability to perform analysis without the probe effect is a significant advance in the maturity and applicability of HMBA tools and plays a vital role in their wider adoption in industry.

Acknowledgements

This work has been supported by the MERASA STREP-FP7 European Project under the grant agreement number 216415.

References

1 The Nexus 5001™Forum. http://www.nexus5001.org, April 2010.

2 T. Austin, E. Larson, and D. Ernst. SimpleScalar: an Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

3 G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02)*, December 2002.

4 G. Bernat, M.J. Newby, and A. Burns. Probabilistic Timing Analysis: an Approach using Copulas. *Journal of Embedded Computing*, 1(2):179–194, November 2005.

5 J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

6 A. Colin and G. Bernat. Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proceedings of the 14th Euromicro Conference of Real-Time Systems (ECRTS'02)*, July 2002.

7 A. Colin and S. M. Petters. Experimental Evaluation of Code Properties for WCET Analysis. In *Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, December 2003.

8 ARM development tools. http://www.arm.com, April 2010.

9 J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, April 2002.

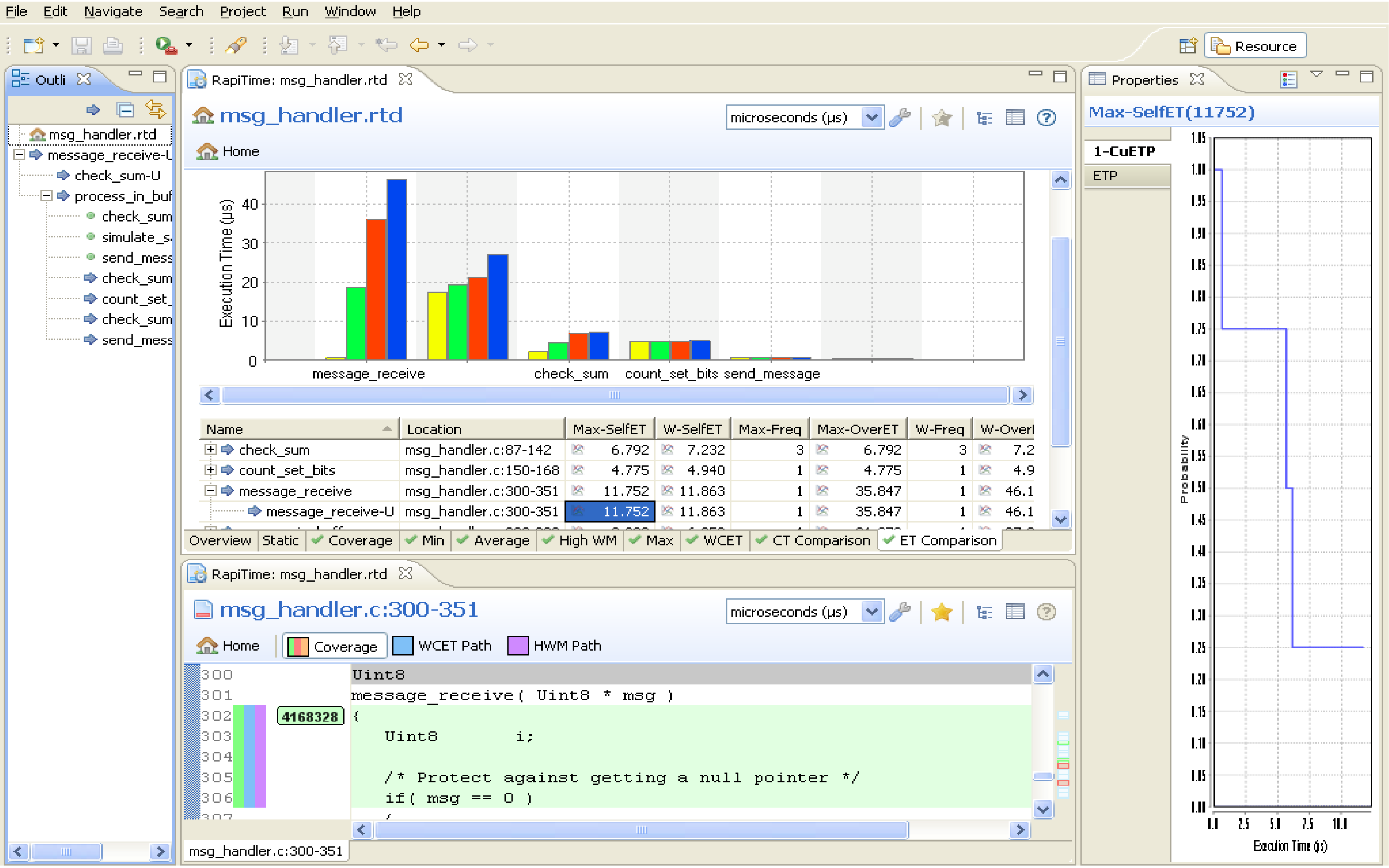
10 T. W. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

11 S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

12 P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, September 1989.

13 P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, July 1997.

14 F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, November 2001.



Name	Location	Max-SelfET	W-SelfET	Max-Freq	Max-OverET	W-Freq	W-Overl
⊞ check_sum	msg_handler.c:87-142	6.792	7.232	3	6.792	3	7.2
⊞ count_set_bits	msg_handler.c:150-168	4.775	4.940	1	4.775	1	4.9
⊟ message_receive	msg_handler.c:300-351	11.752	11.863	1	35.847	1	46.1
message_receive-U	msg_handler.c:300-351	11.752	11.863	1	35.847	1	46.1

```
300  Uint8
301  message_receive( Uint8 * msg )
302  {
303      Uint8       i;
304
305      /* Protect against getting a null pointer */
306      if( msg == 0 )
```

Figure 4 Screenshot of RapiTime Report.

On the Use of Context Information for Precise Measurement-Based Execution Time Estimation*

Stefan Stattelmann[1] and Florian Martin[2]

1 **FZI Forschungszentrum Informatik**
Haid-und-Neu-Str. 10–14, D-76131 Karlsruhe, Germany
stattelmann@fzi.de

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

Abstract

The present paper investigates the influence of the execution history on the precision of measurement-based execution time estimates for embedded software. A new approach to timing analysis is presented which was designed to overcome the problems of existing static and dynamic methods. By partitioning the analyzed programs into easily traceable segments and by precisely controlling run-time measurements with on-chip tracing facilities, the new method is able to preserve information about the execution context of measured execution times. After an adequate number of measurements have been taken, this information can be used to precisely estimate the Worst-Case Execution Time of a program without being overly pessimistic.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.64

1 Introduction

Information about the execution time of programs in embedded systems must be available at several design stages. During the initial phases, a rough estimate of the execution times should be available so that components which fit the expected workload of the system can be chosen. In the final phase of a project, precise execution times must be known in order to verify that the system fulfils all its timing requirements. The increasing complexity of real-time systems makes reasoning about the execution time of embedded software more and more challenging. This particularly holds for the Worst-Case Execution Time (WCET) of a task since it might only occur under rare circumstances which are caused by a nontrivial interaction of system components.

Existing methods for WCET analysis can be divided into static and dynamic methods. Static timing analyses try to determine a safe upper bound for all possible executions of a given program. In contrast, dynamic methods use measurements taken during a finite number of actual executions to determine an estimate of the WCET. On current processor architectures, both methods do not always produce satisfying results. The interaction of performance enhancing features in modern processors makes it very unlikely to observe the worst-case execution during a few test runs. Hence measurement-based analyses might underestimate the WCET considerably. Furthermore, existing methods are often not able to

* This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under grant 01M3088C and by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) funded by the European Commission's 7th Framework Programme on Research, Technological Development and Demonstration under contract 215068.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 64–76
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

represent the performance gain from caches (cache effects) precisely. If only the worst-case execution time is considered for each basic block in a loop body, a performance increase in following iterations due to caching cannot be represented. This can make dynamic estimates very pessimistic, too. Static analyses must use (safe) approximations for the potential states of the analyzed system as the state space can grow very large for sophisticated architectures. These approximations are necessary to make the computation of WCET estimates feasible, but the increase of the reported bounds and the resulting imprecision can restrict their practical use.

This paper presents a context-sensitive analysis of accurate instruction-level measurements which can provide precise worst-case execution time estimates. The notion of context-sensitivity is a well-known concept from static program analysis. It has been shown that the precision of an analysis can be improved significantly if the execution environment is considered. This especially holds if the analysis does not only consider different call sites but also distinct iterations of loops (see [12]), as this allows the consideration of cache effects. Up to now, context information is mainly used in static timing analysis. The results presented in this work suggest that the precision of measurement-based timing analyses, too, can be increased considerably by incorporating context information.

Recent developments in debug hardware technology allow the creation of cycle-accurate traces with a fully programmable on-chip event logic [13]. The increasing availability of these tools for instruction-level measurements and the precise timing information they provide motivate the use of methods from static timing analysis for measurement-based approaches. As dynamic methods for WCET estimation can be adapted to new processor architectures much more easily than the models used in static analyses, this would reduce the initial investment necessary for performing exact timing analyses.

The influence of context information on the precision of execution time estimates is not only interesting for WCET analysis. All forms of execution time inspection on complex architectures might be improved by incorporating context information, even if the worst-case is not (yet) of interest, like during design space exploration in an early development phase.

The remainder of this work will be organized as follows. The next section will list some related approaches for dynamic WCET analysis. Section 3 introduces a new context-sensitive method for measurement-based execution time analysis. In the fourth section, experimental results with this method will be presented. The last section gives a summary of the work and the impact of the results.

2 Related Work

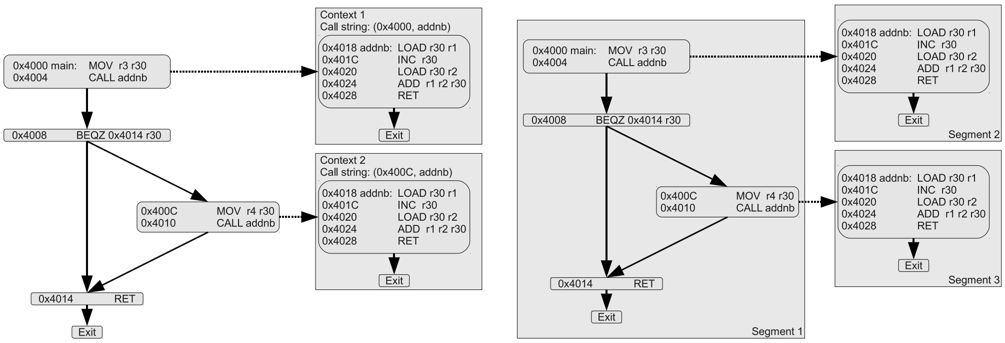
A complete overview of existing methods for WCET analysis can be found in [20]. This section will focus on measurement-based methods. One of the first attempts to consider the execution context of execution times was the *structure-based approach*, a technique for static timing analysis proposed in [16], but it only aimed at a more precise combination of execution times from individual program parts. Similar techniques are still used for measurement-based WCET analysis, e.g. in [5], but they lack the ability to reflect the interaction of individual program parts which is mandatory to represent the influence of the execution history for example due to cache effects. An extension to the structure-based approach which can distinguish execution times of loop iterations is described in [6], but there is no practical evaluation of the effects on the execution time estimates.

To overcome the problem of exhaustive measurements, several solutions have been developed (e.g. [7] and [18]) that try to partition a program into parts which can be measured more easily. These approaches usually assume that the system can be brought into a worst-case state before taking measurements, e.g. by clearing the cache. This assumption may hold on simpler processors, but it is hard to fulfil in complex systems as it might not be clear what this state looks like due to a complex interaction of system components [9]. Modifying the system state can also make the execution time estimates very pessimistic, for example if the cache is cleared too often during the measurements. Further solutions include the automatic generation of input data to enforce a worst-case execution [19, 18] or the probabilistic analysis of measurements [5]. The use of results from a cache analysis to guarantee that a sufficient number of measurements have been performed was described in [14]. An approach based on game theory which can represent varying execution times for different loop iterations is presented in [15]. It is also based on program partitioning, an automatic generation of input data and requires that all loops in the analyzed program can be unrolled completely. In [11] constraint logic programming is used to model context-sensitive execution times based on constraints that are derived from an execution time dependency analysis of program traces.

Although only an incomplete overview of methods for measurement-based timing analysis can be given here, the main concern of existing approaches seems to be to enforce the observation of a worst-case execution. Furthermore, many techniques require that the analyzed program is changed since instrumentation code must be added. In contrast to this, the following section will introduce an approach which aims at the precise combination of a large number of measurements. By using evaluation boards which provide hardware support for controlling the collection of trace data, probe effects are avoided since code instrumentation is not necessary. Furthermore, steering measurements precisely during runtime allows increasing the precision of execution time estimates since cache effects can be represented by distinguishing the execution history of an observed code region.

3 Proposed Method

This section presents a new concept for measurement-based timing analysis. The method works on the interprocedural control flow graph (ICFG) of a program executable and requires measurement hardware that can be controlled by complex trigger conditions. The development of the approach was motivated by the limited size of trace buffer memory which is available in current hardware for on-chip execution time measurements. Measurement hardware which stores traces in an external memory overcomes this problem by sacrificing accuracy. Due to bandwidth constraints these traces only store certain instructions, for example taken branches. Additionally, timestamps for these instructions are often only created when a partial trace is transfered from a small on-chip buffer to the large external memory. Hence deriving the execution time of every single instruction is hardly possible. As a consequence of these limitations, it is not feasible to determine context-sensitive execution times from end-to-end measurements, since it is not possible to create cycle-accurate end-to-end traces for programs of realistic size, i.e. traces containing a timestamp for *every* executed instruction. Instead of using traces of complete program runs, this work investigates the use of the programmable trigger logic in state-of-the-art evaluation boards for embedded processors to create context-sensitive program measurements. Current tracing technology, like the Infineon Multi-Core Debug Solution [13], allows considering the execution history of a program before starting a measurement run. This is achieved by dedicated event logic in the actual hardware which can be used to encode state machines to model the program



(a) ICFG with virtual inlining

(b) Partitioning into segments

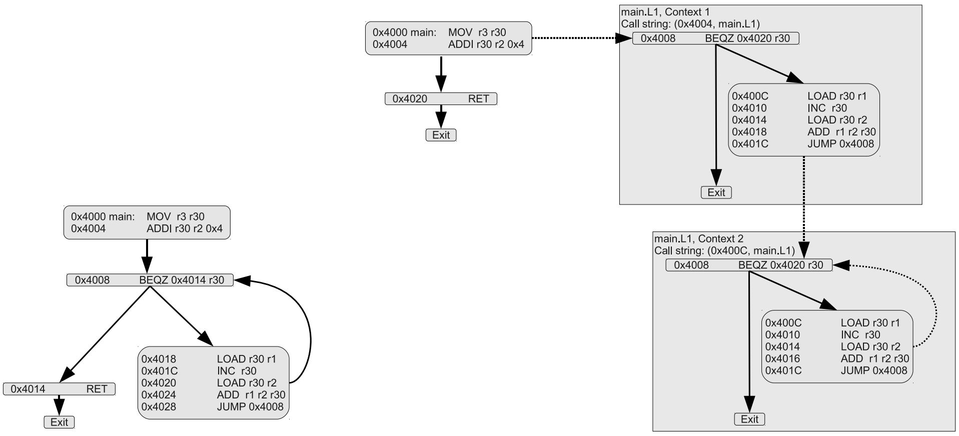
Figure 1 Execution contexts and program partitioning

state. These possibilities motivated the development of an analysis which makes use of this additional logic to generate context-sensitive traces despite the limitations of the trace buffer size. The analysis is divided into several phases:

- Initially, the ICFG is created and partitioned into *program segments* in such a way that every possible run through the segments can be measured with the available trace buffer memory.
- The information gathered during the partitioning phase is used to generate trace automata that will control the measurements.
- Taking measurements requires a sufficiently large number of actual executions of the analyzed program on the target hardware.
- After the measurements have been taken, the context-sensitive timing information for each basic block of the program can be extracted and annotated to the ICFG. Further computations then yield the worst-case path through the ICFG and an estimate of the worst-case execution time of the program.

3.1 Control Flow Extraction

Initially, the control flow is extracted from the program executable and its ICFG is constructed. To represent the execution context of program parts precisely, the concept of *virtual inlining, virtual unrolling* (VIVU) is used [12]. VIVU applies function inlining and loop unrolling on the level of the ICFG. Thus the ICFG can contain several copies of the same basic block for which different execution times can be annotated. To consider the execution history of these duplicates, the control flow graph is extended with additional information that represents the execution context. A *call string* is used to model a routine's execution history. Call strings can be seen as an abstraction of the call stack that would occur during an execution of the program. In this work, a call string will be represented as a sequence of *call string elements*. A Call string element (b, r) is a pair of a basic block b and a routine r which can be called from b. Only *valid* call string elements will be allowed, meaning it must be possible that the last instruction of the basic block b is executed immediately before the first instruction of routine r. For the entry routine of the analyzed task (e.g. *main* in a standard C program) there is no execution history as the execution is started by calling the respective routine. This context is described by the empty call string ϵ. It will be omitted in the following examples. The intuition behind this representation of an execution context is that whenever a routine is called, the call string is extended with another element to



(a) Original CFG **(b)** First iteration unrolled

Figure 2 Extension of the CFG by virtual unrolling

describe the context of the function body. Therefore extending the call string works similar to extending the call stack during program execution. Since the execution history of a routine can be very complex, its call string representation might become very long. In order to achieve a more compact representation of execution contexts, the maximal length of call strings will be bounded by a constant $k \in \mathbb{N}_0$. For call strings which describe a valid execution but exceed the maximal length, only the last k call string elements will be used to describe the context. In the following examples a call string length of one will be used.

Figure 1a depicts an example of virtual inlining in an ICFG by duplicating routine bodies for every call site. Intraprocedural edges are drawn with solid lines, while the edges describing a function call are represented by dashed lines. Routines are not explicitly highlighted in the ICFG, but every routine is assumed to have a unique entry node, which is the target of the call edges, and an artificial exit node through which the routine must be left. The effect of virtual unrolling is shown in in Figure 2. Virtual unrolling also extends the ICFG by duplicating nodes. Additional precision is gained by extracting loops from their parent routine and treating them like recursive routines. This allows a more precise classification of the execution history than a simple calling context when searching for the WCET path through the program, since varying execution times in different loop iterations can be represented independently from the parent routine.

3.2 Program Partitioning

To cope with the limited memory for trace data, the ICFG is partitioned into *program segments*. These segments consist of a start and an end node in the graph which must fulfil the condition that the longest path in terms of executed instructions (not execution time) between them is smaller than or equal to the number of instructions for which timestamps can be stored in the trace buffer. Additionally, both nodes must be part of the same execution context. Segments can either include all nodes which lie on the interprocedural paths between the start and the end node or they can be restricted to the nodes on the intraprocedural paths. By excluding calls to other routines, the size of a program segment can be reduced, but information about the execution context can be preserved in the traces. After this partitioning, each node of the ICFG is covered by at least one program segment and it suffices to perform measurements for individual segments to determine context-sensitive execution times for every basic block.

The program from Figure 1a will be used to illustrate the concept of program segments. Assume the program is to be traced with a trace buffer which can hold timestamps for at most 6 instructions. In order to extract cycle-accurate and context-sensitive timing information, at least 3 program segments are necessary. Each of these segments is measured individually and the results are combined during a post-processing phase. Figure 1b illustrates one possible partitioning. In this example, a separate segment is created for the body of the routine *addnb* at each call site. Additionally, the segment for the top-level routine *main* is assumed to be measured without the routine it calls. This assumption makes it possible to handle the limited trace buffer. However, to fulfil this assumption during an actual measurement run, it must be possible to trigger the measurement hardware precisely.

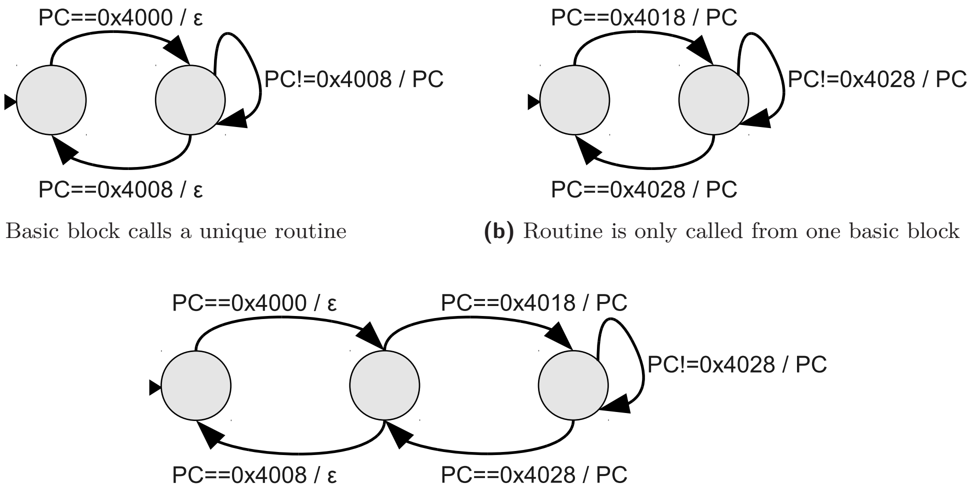
3.3 Trace Automata

For each program segment, a *trace automaton* is generated to control the measurement runs of the respective segment. These state machines encode the conditions which describe the execution history of each segment, i.e. which instructions must have been executed before the execution context of the observed program run matches the program segment. Monitoring the execution of the program before starting a measurement allows preserving information about the execution context even if the trace does not contain the complete execution history of the analyzed code regions. The automata are constructed from the execution history of the respective segments, which is described by a sequence of call sites and loop entries (call string). This abstract description of trace preconditions is translated to the event logic of the evaluation board using a software debugger [4] which is then used to collect measurement data.

For the description of an execution context, each element of a call string describes two conditions in terms of executed instructions: the call instruction in the call block must be executed immediately before the first instruction of the called routine. Additionally, the sequence of the elements constrains the order of these conditions, i.e. the order of the calls. In principle, they can be directly translated to a trace automaton which changes its state depending on whether the correct routines are called at the appropriate call sites. But since most call sites call exactly one routine, the automata created by this strategy are not minimal. On the other hand, there might be program segments which have a common call string, but lie in a different instruction address range (e.g. if a routine gets partitioned into two segments). Hence it is not sufficient to consider only the context description when constructing trace automata.

To generate a trace automaton for measuring a program segment, the first step is to create states and transitions which correspond to the constraints described by the call string representation of the segment's execution context. After that, states must be added to express which instructions on the paths through the segment should be traced. The complete approach proceeds as follows:

Initially, the automaton has a single state, no transitions and generates no output. Then, at least one state for each element of the call string is added to the automaton. How many states are added depends on the properties of the call site described by the string element. If the call described by the element has only one possible destination, it suffices to use the address of the call block as condition for the transition to the next state. Similarly, if this is not the case but the destination routine is only called at this call site, it is enough to add a single state which is entered as soon as the entry address of the respective routine is



(a) Basic block calls a unique routine

(b) Routine is only called from one basic block

(c) Several callers and callees

Figure 3 Translation of a call string element

encountered. For call string elements which fulfil neither of the conditions, both states have to be added to the trace automaton to model the requirements for the execution history described by the call string elements. These three cases are illustrated in Figure 3 for the one-element call string (*0x4000*, *addnb*) and the routine *addnb* from Figure 1a which starts at the memory location *0x4018* and returns at address *0x4028*.

Finally, the state for actually storing trace data is added. For program segments which cover all routines that are called on the paths through the segment, this can be done by adding a single state which is entered as soon as the start block is entered and left when the end block is left. In this state, all instructions which will be executed will also be stored in the trace buffer. For segments which exclude called routines, things are slightly more complicated and an additional state gets necessary. The tracing state is constructed as before, but the additional state is entered when calls are executed within the segment (i.e. when the address interval for the segment is left) and no trace data will be generated while the automaton is in this state. Note that no extra state for storing trace data is necessary if the program segments covers a complete routine and all its calls. In this case, tracing can start as soon as the addresses for all call string elements of the execution context have been processed by the automaton. This is also illustrated in Figure 3.

3.4 Trace Data Generation

Taking measurements requires a sufficiently large number of actual executions of the analyzed program on the target hardware. The approach relies on the assumption that all worst-case execution times of each basic block in every execution context were observed during the measurements to produce a safe estimate. As the program under consideration is not modified in any way, measurements should be taken under realistic conditions to produce execution time estimates that match the expected workload. Using typical inputs during a large number of measurements should result in estimates close to the actual WCET. Controlling the generation of trace data with state machines offers the advantage that the measurement logic can wait for code region which are executed rarely before triggering the trace generation. Since this process can be automated, achieving a sufficient level of code coverage is facilitated, but not guaranteed.

3.5 Timing Extraction

After a set of traces has been generated for each of the segments into which the program of interest was partitioned, the maximal execution time for each basic block is extracted from the measurements. As the traces are assumed to be cycle-accurate, this is a straightforward process since every instruction which gets executed during a measurement run must also be contained in the respective trace. Additionally, the traces must contain a (relative) timestamp for each instruction. Since tracing is controlled by (an implementation of) a trace automaton, the precise execution context of the trace data is known. Hence the execution time for each basic block can be extracted from a trace by simply going through the trace and the ICFG in parallel. Whenever a new basic block is entered in the trace, the respective node must be found in the ICFG. Depending on the type of program segment which is annotated, this search for a successor must be carried out on the whole ICFG or just within the current routine, i.e. without following call edges. The execution time of a basic block is determined by subtracting the timestamp of its first instruction from the timestamp of the first instruction of its successor block. As the context in which a trace is generated is preserved while creating the measurements, basic block execution times from the trace are only annotated to those nodes with matching context. In case of virtual inlining, this means that execution times are only annotated to those nodes in the ICFG at the correct call site, but not to the nodes in other contexts (although these nodes represent the same basic blocks on assembly level). Depending on the level of unrolling, the execution times of nodes within a loop can also be assigned to distinct iterations. Hence the worst-case execution time of nodes in the first iteration of a loop, which might generate many misses in the instruction cache, can be easily separated from the remaining iterations. Further iterations are usually not expected to suffer the same performance penalty from cache misses. By duplicating these nodes, the WCET estimates get more precise compared to approaches which cannot make this distinction. In contrast to the method presented in [11], no additional processing of the traces has to be performed to derive these dependencies between basic block execution times. Additionally, the worst-case execution time of the whole program can still be computed by implicit path enumeration [10] and there is no need to resolve additional execution time constraints using constraint logic programming.

For each node in the ICFG which was covered by a trace, this provides the execution time for this particular run. Under the assumption that all local worst-cases were observed during the measurements, meaning that the worst-case execution time of each node is covered by at least one of the traces, the maximum from all of the execution times equals the worst-case execution time. All nodes in the ICFG which never occurred in one of the traces are assumed to be never executed during *any* execution of the program. If a *sufficiently large* number of measurements has been taken under *realistic conditions*, taking the maximum of the measured execution times for each node is likely to provide the worst-case execution time or at least a *realistic estimate* of it. Nevertheless, the presented work provides no means to enforce these conditions.

3.6 Cache Analysis

The design of the proposed method allows an easy integration of static analyses to make the measurement phase more efficient. To demonstrate this, the cache analysis described in [8]

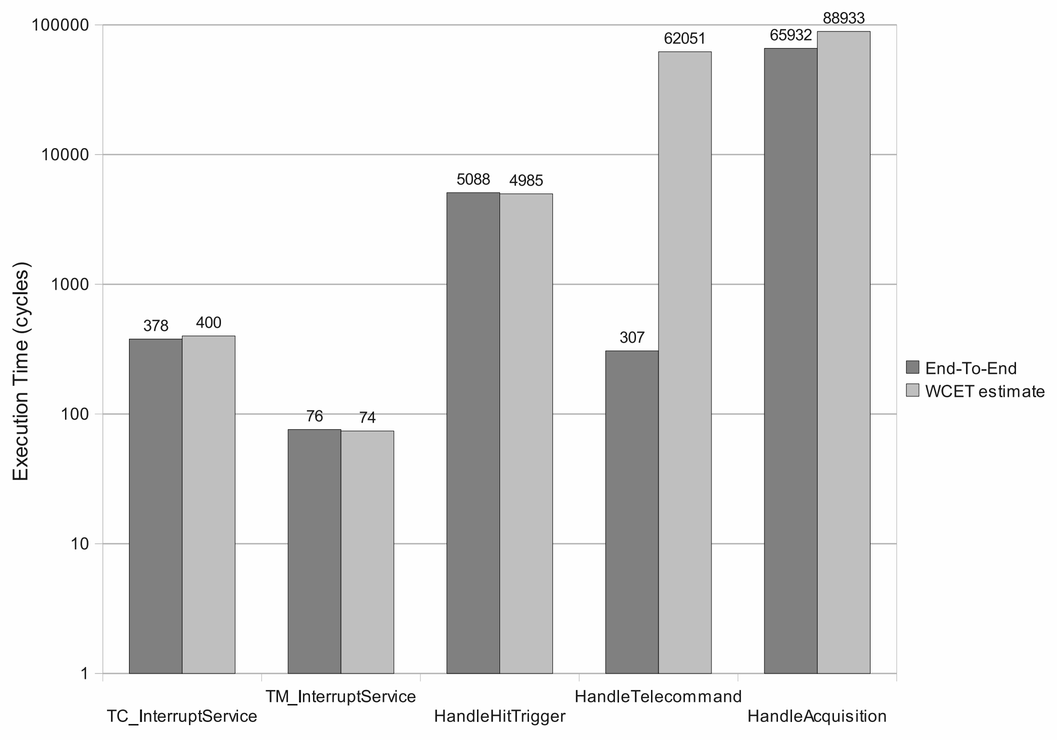


Figure 4 Comparison of context-sensitive and end-to-end measurements

has been adapted to classify the instruction cache behavior of different execution contexts. This classification is achieved by comparing the number of potential and guaranteed cache hits and misses for different execution contexts of a basic block. If two program segments from the same code region, but with a different execution history, will exhibit a (roughly) identical cache behavior, they will not be distinguished during the measurements. By joining some of the execution contexts generated by the VIVU approach, the number of measurements can be reduced without influencing the precision of the WCET estimates. A detailed description of this optimization is beyond the scope of this paper, but further details can be found in [17].

4 Experimental Results

The proposed method was implemented as an extension of the AbsInt aiT WCET Analyzer [1] and tested with an Infineon TriCore TC1797ED evaluation board. Several common WCET benchmarks could be successfully analyzed, in particular programs from the Mälardalen WCET Benchmark Suite [3] and the DEBIE-1 benchmark [2], which is an adapted satellite control application. No changes to the hardware state or the analyzed software were performed during the experiments. The programs were simply loaded into the flash memory of the evaluation board and then measured several times successively. Input data for the programs did not have to be generated as this was already handled by the benchmarks.

For an initial test, the estimates provided by the implementation were compared to WCET estimates based on a number of simple end-to-end measurements. The result of this comparison is shown in Figure 4. Estimating the worst-case execution time of programs

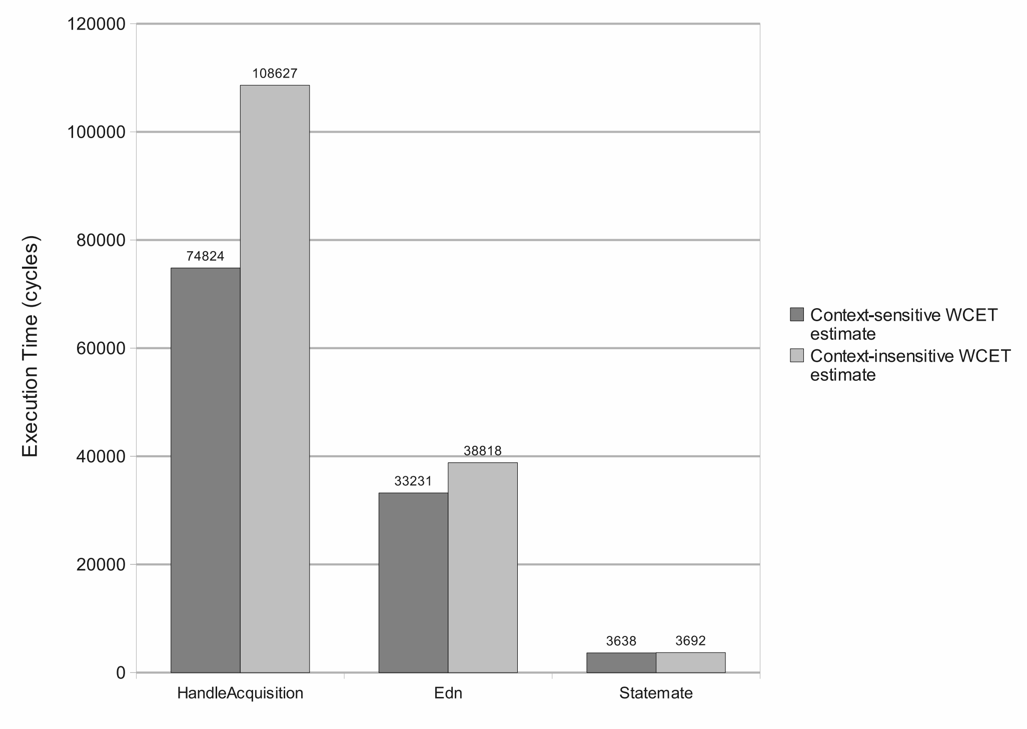


Figure 5 Improvement of WCET estimates through context information

based on measurements is problematic as it usually cannot be guaranteed that the worst-case has been covered. This was demonstrated by the test case *HandleTelecommand* from the DEBIE-1 benchmark: though a considerable effort was made for the measurements, the observed end-to-end execution times were considerably smaller than the context-sensitive WCET estimates. Manual examination of the traces showed that some routines which were on the WCET path reported by the context-sensitive approach were never executed during the end-to-end measurements. Hence this test case showed that for programs which rarely execute the routines which are responsible for the worst-case execution, the presented approach is superior to simpler methods. The program partitioning and the precise control over the measurement runs allows the measurement hardware to wait for these rarely executed program parts before starting the actual trace. Nonetheless, the prototype implementation reported some WCET estimates which were smaller than the maximal execution time observed during the end-to-end measurements. One reason for this is that the measurement hardware sometimes did not start the traces immediately after they were triggered. As a result of these delays, some basic blocks were never completely covered by the measurements and thus the execution time was underestimated. As the number of measurements which were taken during the experiments was relatively small, insufficient coverage of critical program parts is another potential cause of the underestimation.

The effect of context information for measurement-based execution time estimation was investigated by using the same set of trace data with and without the consideration of the execution history (Figure 5). A smaller number of measurements was performed for these experiments than for the previous ones as the focus was not on precisely estimating the WCET (i.e. covering *all* local worst-cases), but on investigating the effect of context information. For this reason, some results presented in Figure 5 differ slightly from previous estimates. The context-insensitive analysis uses the maximal execution time of each basic block found in the traces and annotates this value to every copy of the respective basic block in the ICFG. On the other hand, the context-sensitive analysis was able to annotate smaller execution times to some of the basic block instances since it is able to preserve information about the execution history, e.g. by distinguishing the first from all remaining iterations of a loop. For two out of three test cases, the context-sensitive approach seems to be able to represent cache effects more precisely. Hence, smaller WCET estimates are reported. This effect could not be observed for the smallest of the test cases, probably since the execution time of the program does not benefit from caches due to its linear structure. The results of this comparison suggest that the difference between a context-sensitive and a context-insensitive analysis can be substantial. By increasing the number of measurement runs, this effect can only be intensified, as for every increase in the context-sensitive estimate, the context-insensitive estimate must grow as well. Thus the execution context of execution time measurements should be preserved whenever possible. If this is not done, cache effects cannot be determined correctly, which is why a context-insensitive evaluation might introduce a severe amount of pessimism to the execution time estimates, which renders them less precise.

5 Conclusion

This work proposed a new approach to measurement-based timing analysis which makes use of techniques from static program analysis. The results obtained during the experiments show that state-of-the-art measurement hardware can be used to determine WCET estimates automatically. To get precise results, a large number of measurements should be performed since the method relies on the assumption that the local worst-case for each basic block was observed during the measurements. Although the new approach seems to be more robust and more precise than existing methods for measurement-based timing analysis, it does not overcome their inherent problems, like the dependence on input data. However, controlling the collection of trace data precisely allows weakening the influence of these problems to the WCET estimate, e.g. because it is now possible to facilitate measurements within program parts or execution contexts which are executed very rarely. While the precise control of trace data generation makes it more likely that local worst-case executions can be observed, the use of context information allows the precise combination of partial execution times. This makes the calculated WCET estimates less pessimistic.

The outcome of the experiments also shows that only measuring each basic block often enough, which is the prevailing paradigm for measurement-based timing analysis, is not enough to determine precise execution time estimates as the execution history might have a significant influence on them. For static timing analysis this is a well-known fact, but the presented results suggest that all techniques for reasoning about software execution times on complex hardware can benefit from the use of context information. This includes measurement-based execution time analysis as well as techniques for execution time estimation in a design space exploration or simulation environment.

As a complex event logic for trace data generation is not always available, measurement-based methods for WCET analysis could try to reconstruct context information from trace data instead of controlling its creation. This should still improve the precision of the estimates, but would also work with simpler measurement facilities. Extracting a context-sensitive worst-case execution time for each basic block from the trace data has the additional benefit that only one value has to be stored for every execution context. As more traces are generated, these values are only updated if a longer execution time has been observed. All execution times which are below the worst-case can be discarded without influencing the final result. This allows the efficient parallelization of trace data generation and timing extraction. Additionally, this approach can also help to reduce the tremendous storage requirements which measurement-based methods for WCET analysis often have.

Acknowledgements The authors would like to thank Christian Ferdinand for fruitful discussions about the content of this paper.

References

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

2 Debie-1 WCET Benchmark. http://www.mrtc.mdh.se/projects/WCC08/doku.php.

3 Mälardalen WCET Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet.

4 pls Development Tools Universal Debug Engine (UDE). http://www.pls-mc.com.

5 Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. Technical report, Department of Computer Science, University of York, February 2003.

6 Adam Betts and Guillem Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 558–565. IEEE Computer Society, 2006.

7 Jean-François Deverge and Isabelle Puaut. Safe Measurement-Based WCET Estimation. In Reinhard Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

8 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

9 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

10 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995. ACM.

11 Amine Marref and Guillem Bernat. Towards Predicated WCET Analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

12 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC '98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94, Berlin, 1998. Springer.

13 Albrecht Mayer and Frank Hellwig. System Performance Optimization Methodology for Infineon's 32-bit Automotive Microcontroller Architecture. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 962–966. ACM, 2008.

14 Stefan Schaefer, Bernhard Scholz, Stefan M. Petters, and Gernot Heiser. Static Analysis Support for Measurement-Based WCET Analysis. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Work-in-Progress Session*, 2006.

15 Sanjit A. Seshia and Alexander Rakhlin. Game-Theoretic Timing Analysis. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 575–582, Piscataway, NJ, USA, 2008. IEEE Press.

16 Alan Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15:875–889, 1989.

17 Stefan Stattelmann. Precise Measurement-Based Worst-Case Execution Time Estimation. Master's thesis, Saarland University, September 2009.

18 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Timing Analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Oct. 2008.

19 Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar. 2005.

20 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem — Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

A Code Policy Guaranteeing Fully Automated Path Analysis

Benedikt Huber[1] and Peter Puschner[1]

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

Abstract

Calculating the worst-case execution time (WCET) of real-time tasks is still a tedious job. Programmers are required to provide additional information on the program flow, analyzing subtle, context dependent loop bounds manually. In this paper, we propose to restrict written and generated code to the class of programs with input-data independent loop counters. The proposed policy builds on the ideas of single-path code, but only requires partial input-data independence. It is always possible to find precise loop bounds for these programs, using an efficient variant of abstract execution. The systematic construction of tasks following the policy is facilitated by embedding knowledge on input-data dependence in function interfaces and types. Several algorithms and benchmarks are analyzed to show that this restriction is indeed a good candidate for removing the need for manual annotations.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.77

1 Introduction

Worst-Case Execution Time (WCET) analysis is concerned with determining an upper bound for the time needed to execute a task or procedure on a particular architecture. It is a necessary prerequisite for verifying that a system meets its timing specification. A widely used approach for WCET analysis is to analyze the set of execution paths and the timing of instruction sequences separately. The latter also includes the analysis of hardware components with global state, such as instruction and data caches or pipelines. Finally, the results of both the high-level and low-level analysis are fed into a solver, which computes the maximum execution time [12].

Flow facts are constraints that describe restrictions on the set of possible execution sequences. The path analysis has to determine a set of finite execution sequences as an overapproximation to all possible execution paths. As each considered execution path has to be finite, it is necessary to bound loop iteration counts and recursion depths. Together with information about the target of indirect jumps, these bounds are sufficient and necessary to derive some WCET bound. Loop bounds are either detected using an automated loop bound analysis, or are specified by the programmer through annotations.

WCET analysis is primarily concerned with the execution time of machine code on one particular architecture. Therefore, knowledge about the execution paths of a task is only useful if it can be mapped to machine code in a reliable way. Due to compiler optimizations, source code annotations are difficult to map to machine code. Annotations on the assembler level are difficult and tedious to write, and cannot be reused when the program is recompiled.

One solution to this dilemma is to have both a language that allows to integrate, test and verify flow information, and compilers that are aware of these flow facts, and transform them into flow facts on the machine code level [7, 3]. While highly desirable, flow-fact aware compilers are still rare. More importantly, source code annotations tend to be error prone, and often stay untested.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 77–88
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Another potential solution to eliminate the need for machine code level annotations is to derive all necessary flow facts automatically using an automated loop bound analysis. A variety of dataflow techniques has been proposed and implemented (e.g. [8]), as well as more expensive methods such as symbolic model checking. The problem with these techniques is that it is hard to predict whether they will find all necessary flow facts. In the fields of compiler optimization and testing, where most techniques were developed, a successful analysis is useful, but not crucial for correctness. In the WCET analysis domain, however, failing to derive loop bounds requires the user to resort to manual annotations.

We think that being unable to predict the success of loop bound analysis is a major obstacle for building systems that do not depend on user annotations. While for arbitrary code, there is little hope to find all necessary flow facts automatically, the situation is different if the implementation has to follow certain rules. Single path code [9] is characterized by the fact that there are no input-data dependent decisions and consequently only one possible execution path. For single-path code, all flow facts can be obtained by simply executing the program and recording the execution trace. Moreover, it is possible to transform all programs with known loop bounds to single-path form [10]. However, the single path policy is perceived to be too restrictive, as it does not allow any input-data dependent control flow at all.

In this paper, we propose a formal code policy less restrictive than single path, which does not ban the use of all input-data dependent control flow decisions. It e.g. allows to use state machines, without the need to transform the corresponding dispatch table to non-branching code, as required by the single-path concept. The policy is still sufficiently restrictive to guarantee that a simple and efficient form of abstract execution [4] will find the flow facts necessary for bounding the execution time.

Intuitively, the policy presented in this paper requires that all loops have at least one exit condition independent of input data. We present a formal definition of input-data dependence, and automatically classify input-data independent decisions. To this end, we use a program analysis technique which is similar to the one given in [5], but provides a more liberal definition of input-data dependence. Furthermore, the notion of input-data independence can be included in the application programming interface (API), facilitating a systematic construction of programs known to be analyzable.

As an example, compare the two implementations of binary search given in Listing 1a and Listing 1b, assuming that the size of the array is known. Both implementations have a similar performance, but different characteristics when it comes to loop bound analysis. The classic implementation in Listing 1a requires a relatively complicated proof to establish the loop bound. In contrast, it is easy to calculate the loop bound for the implementation in Listing 1b. If the size of the array is input-data independent, the second implementation agrees with the proposed policy. The single path implementation, which allows to calculate the exact number of iterations for a given array size, is shown in Listing 1c.

The crucial question deciding the acceptance of this methodology is whether it is too restrictive or not. To this end, we argue that many algorithms we believe to be useful in typical hard real-time systems can indeed be designed to follow the policy in a natural way. Furthermore, it is possible to transform manually annotated loops into loops with an input-data independent exit condition. While this does not solve correctness and maintainability problems of annotations, it shows that all tasks can be written in a way adhering to the policy.

```
int bsearch_std(int arr[], int N, int key)
{
  int lb = 0;
  int ub = N - 1;
  while (lb <= ub)
  {
    int m = (lb + ub) >>> 1; /* unsigned shift */
    if (arr[m] < key)      lb = m+1;
    else if (arr[m] > key) ub = m-1;
    else                   return m;
  }
  return -1;
}
```

(a) Dependent loop counter

```
int bsearch_idi(int arr[], int N, int key)
{
  int base = 0;

  for (int lim = N; lim > 0; lim >>= 1)
  {
    int p = base + (lim >> 1);
    if (key > arr[p]) base = p + (lim&1);
    else if (key == arr[p]) return p;
  }

  return -1;
}
```

(b) Independent loop counter

```
int bsearch_sp(int arr[], int N, int key)
{
  int base = 0;
  int r = -1;

  for (int lim = N; lim > 0; lim >>= 1)
  {
    int p = base + (lim >> 1);
    if (key > arr[p])  base = p + (lim&1)
    if (key == arr[p]) r    = p;
  }
  return r;
}
```

(c) Single path

Listing 1 Different Implementations of Binary Search

Outline

In Section 2, we discuss the notion of input-data dependence, and give a formal definition, which is easy to check both manually and automatically. Section 3 defines the class of tasks with input-data independent loop counters, and shows how to extend function interfaces and the type system to capture the intended input-data independence of variables. In Section 4, we propose a simple and efficient abstract execution framework to extract precise loop bounds. In Section 5, examples of algorithms and real-time benchmarks are evaluated with respect to the policy. Finally, Section 6 discusses future work and concludes the paper.

2 Input Data Independence

To analyze the WCET of a function, it is in general necessary to assume additional restrictions on the initial state of variables. For example, the execution time of a function performing a binary search depends at least on the size of the input array. The tasks scheduled in hard real-time systems, however, must always have an absolute WCET bound, which is provided as input to the scheduling algorithm. The policy presented in this paper requires certain variables to be input-data independent with respect to real-time tasks.

An expression is input-data independent if its value at a specific instruction of one execution trace does not depend on the environment. This includes dependencies on sensor values, timers and other values obtained from outside the system, as well as dependencies on other tasks or the runtime system, for example due to shared variables.

This would be the ideal notion of input-data independence, but it is intractable to check automatically. For example, if an arbitrary function $f(x)$ returns a constant independent of its parameter x, then $f(x)$ is input-data independent even if x is a value obtained from the environment. But deciding automatically whether the result of some arbitrary function has a dependency on one of its inputs is not possible in general. We do not want to be too restrictive either: An expression should not be unconditionally classified as being input-data dependent only because one decision on the path to the corresponding instruction was.

For these reasons, we define input data dependence in terms of dataflow equations, which can be easily checked by machines and are still comprehensible by humans.

There is already a close correspondence between data dependencies in static single assignment (SSA) form [1] and input data dependencies. In SSA form, the dependencies between the uses of a variable and its definition are made explicit by subscripting variables with an index reflecting their definition site, and adding dedicated statements for merging reaching definitions. Dependencies due to control flow decisions and due to the mutation of fields or array elements are not captured by SSA though.

In Figure 1, the source language for the analysis is defined. The statement $v :=$ `read` assigns v to a statically unknown value obtained from an interaction with the environment. The language includes support for reading and writing elements of arrays and fields of composite types. Variable definitions are in SSA form, i.e. each variable only appears once on the left-hand side of an assignment, and definitions reaching an use site are explicitly merged by so called ϕ functions.

We now define input data dependence in terms of an operator $\mathcal{A}$, which maps each statement to one or more dataflow equations. Following [5], the domain of the analysis is the semilattice $\{ID, IDI\}$. A variable v_i is input-data independent if there is a solution to the data-flow equations listed in Figure 2 with $v_i = IDI$, and input-data dependent otherwise. If a variable depends on two or more other variables, it is input-data independent only if all variables it depends on are. Therefore we define $ID \sqcap IDI = IDI \sqcap ID = ID$.

$v := c$	Assign v to an integer constant
$v := \texttt{read}$	Assign v to a value obtained from the environment
$v := v_1 \circ v_2$	Assign v to $v_1 \circ v_2$, with $\circ \in \{+, -, *, <, \leq, =, \texttt{AND}, \texttt{OR}, \texttt{XOR}\}$
$v := v_1[v_2]$	Assign v to the element at position v_2 of the array v_1
$v[v_1] := v_2$	Set element v_1 of the array v to v_2
$v := v_1.F$	v is assigned to the field F of of v_1
$v.F := v_1$	The field $v.F$ is assigned to v_1
$v := \phi(v_1, \ldots, v_n)$	v is the reaching definition from the set $\{v_1, \ldots, v_n\}$
$\texttt{bz}\ v$	Conditional branch, following the “true” edge if $v = 0$

Figure 1 The input language for input-data dependence analysis

The equations dealing with constants, environment interaction and binary operators are straightforward. Dependencies between control-flow *decisions* and ϕ functions are defined in terms of decision branches. A decision branch of $y := \phi(x_1, \ldots, x_n)$ is a conditional branch $\texttt{bz}\ v_c$, which has a direct influence which definition of x will reach the merge point defining y. If $\texttt{bz}\ v_c$ has an influence which definition reaches $y := \phi(x_1, \ldots, x_n)$, y not only depends on $\{x_1, \ldots, x_n\}$, but also on the condition variable v_c.

Formally, $\texttt{bz}\ v_c$ is a decision branch if there are two paths p_1 and p_2 starting at $\texttt{bz}\ v_c$ and a variable x_i such that (a) p_1 and p_2 only have the first statement ($\texttt{bz}\ v_c$) in common (b) p_1 includes the assignment to x_i (c) p_2 does not include the assignment to x_i (d) the last statement of p_2 is the merge point $y := \phi(x_1, \ldots, x_n)$. To ensure this definition is correct in the presence of loops, we require all code to be in Loop-Closed SSA form. The set of all decision branches of $y := \phi(x_1, \ldots, x_n)$ is denoted by $\mathcal{D}(y)$. The resulting dependencies are reflected in the equations for *phi* functions in Figure 2.

Dependencies due to the mutation of array elements or fields usually require an alias analysis, determining which statements might influence which fields. A straight-forward type-based alias analysis [2] is not suitable for arrays, as arrays with the same element type may be used for both static and dynamic data. Store-based alias analyses distinguish data based on the memory address or the allocation site. As they track the set of all memory locations a variable may point to, the outcome of these analyses is difficult to predict for humans.

Therefore, a type attribute `IDI` is introduced, that distinguishes input-data independent and input-data dependent array types. This is similar to the `const` attribute in C. If v is declared to be input-data independent, we write $\mathit{idtype}(v) = \mathit{IDI}$, otherwise $\mathit{idtype}(v) = \mathit{ID}$. In an assignment $v = v_1[v_2]$, v is input data independent if v_1 and v_2 are input-data independent, and $\mathit{idtype}(v_1) = \mathit{IDI}$. Assignments to arrays need to be type checked. It is required that in an assignment $v[v_1] = v_2$ both v_1 and v_2 are input-data independent if $\mathit{idtype}(v) = \mathit{IDI}$.

For fields of composite types, we need a more fine grained distinction than for arrays, where either all or no elements are declared to be input-data independent. Consider e.g. a datatype for resizable vectors, consisting of three fields, one for the data in the vector, one for the vector’s size and one for its maximum capacity. While some algorithms may require the maximum capacity to be input-data independent, clearly neither the size nor the internal data field have to be.

As it is usually possible to define different composite data types for different purposes, fields of a composite datatype are explicitly declared as being input data independent. In Figure 2, we write $\mathit{idtype}(v.F) = \mathit{IDI}$ to denote that field F of variable v is declared to

$$
\begin{array}{lcl}
\mathcal{A}[v := c] & \equiv & v = IDI \\
\mathcal{A}[v := \texttt{read}] & \equiv & v = ID \\
\mathcal{A}[v := v_1 \circ v_2] & \equiv & v = v_1 \sqcap v_2 \\
\mathcal{A}[v := v_1[v_2]] & \equiv & v = v_1 \sqcap v_2 \quad \text{if } \mathit{idtype}(v_1) = IDI \\
 & & v = ID \quad \text{otherwise} \\
\mathcal{A}[v := v_1.F] & \equiv & v = v_1 \quad \text{if } \mathit{idtype}(v_1.F) = IDI \\
 & & v = ID \quad \text{otherwise} \\
\mathcal{A}[v := \phi(v_1, \ldots, v_n)] & \equiv & v = \bigsqcap v_1, \ldots, v_n \quad \sqcap \quad \bigsqcap_{\texttt{bz}\ c_i \in \mathcal{D}(v)} c_i
\end{array}
$$

Figure 2 Dataflow equations for input-data dependence analysis (without type checking)

be input-data independent. Similar to arrays, a type checker has to ensure that $v.F$ is not declared being input data independent if the right hand side of the assignment $v.F = v_1$ is not.

The example in Figure 3 illustrates the concepts presented in this section. It consists of two loops with an input-data independent loop counter, and a conditional branch, whose condition is input-data dependent. In this example, bz b_2 is the only decision branch, deciding whether r_2 or r_3 reaches $r_5 = \phi(r_2, r_3)$. Therefore, r_5 depends on the condition variable b_2, and is input-data dependent.

3 Input-data Independent Loop Counters

In this section, we will define the class of tasks with input-data independent loop counters. We restrict ourselves to reducible loops [6], i.e., loops with a unique entry node, called the loop header. A conditional branch within the loop is a decision branch for this loop, when there is one outgoing edge that exits the loop.

▶ **Definition 1.** A task has *input-data independent loop counters*, if each loop has at least one input-data independent decision branch, which will eventually terminate the loop. A conditional branch bz v is input-data independent if its condition variable is classified as IDI.

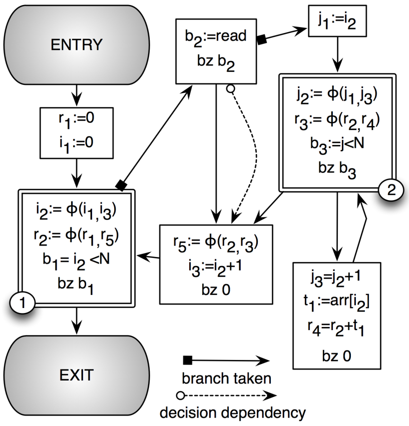
This definition captures those tasks whose loop iteration counts can still be determined after removing all input-data dependent variables. The example in Figure 3 has two loop decision branches, bz b_1 for the outer loop, and bz b_3 for the inner loop. As both are input-data independent, dsum indeed has input-data independent loop counters.

One important goal driving the definition above is that it should be possible to systematically construct tasks with input-data independent loop counters. To this end, the notion of input-data independence is included in the interface definition of functions. The interface specification of a function includes the set of parameters that need to be input-data independent. The caller of the function has to ensure that those parameters which need to be input-data independent indeed are, every time the function is called. In the example, the caller of dsum has to ensure N is input-data independent. Using this assumption, the analysis can prove that the function has input-data independent loop counters locally.

For composite data types and arrays, input-data independence is specified at the definition site. For example, the library provider specifies that the capacity field of a resizable vector always has to be input-data independent. The type system then has to check that no input-data dependent values are assigned to that field.

```
//@precondition: N = IDI
int dsum(int arr[], int N)
{
  int r=0;
  int i=0;
  while(i < N) {
    int c = read();
    if(c) {
      int j = i;
      while(j++ < N) {
        r+=arr[i];
      }
    }
    i=i+1;
  }
  return r;
}
```

Listing 1 Source Code for the Input-Data Dependence Analysis Example



(a) SSA Flow Graph

$$\begin{array}{rcl}
i_1 & = & IDI \\
i_2 & = & i_1 \sqcap i_3 \\
i_3 & = & i_2 \\
\hline
j_1 & = & i_2 \\
j_2 & = & j_1 \sqcap j_3 \\
j_3 & = & j_2 \\
\hline
N & = & IDI \\
t_1 & = & idtype(arr) \\
\hline
b_1 & = & i_2 \sqcap N \\
b_2 & = & ID \\
b_3 & = & j_2 \sqcap N \\
\hline
r_1 & = & IDI \\
r_2 & = & r_1 \sqcap r_5 \\
r_3 & = & r_2 \sqcap r_4 \\
r_4 & = & r_2 \sqcap t_1 \\
r_5 & = & (r_2 \sqcap r_3) \sqcap b_2
\end{array}$$

(b) DF Equations

$$\begin{array}{rcl}
i_1 & = & IDI \\
i_2 & = & IDI \\
i_3 & = & IDI \\
\hline
j_1 & = & IDI \\
j_2 & = & IDI \\
j_3 & = & IDI \\
\hline
N & = & IDI \\
t_1 & = & idtype(arr) \\
\hline
b_1 & = & IDI \\
b_2 & = & ID \\
b_3 & = & IDI \\
\hline
r_1 & = & IDI \\
r_2 & = & ID \\
r_3 & = & ID \\
r_4 & = & ID \\
r_5 & = & ID
\end{array}$$

(c) Solution

Figure 3 Example of the Input-Data Dependence Analysis

4 Generating Flow Facts by Abstract Execution

In this section, we will demonstrate an efficient way to derive all necessary flow facts for tasks with input-data independent loop counters.

For single-path code, all flow facts can be derived automatically by executing the task, and recording the instruction trace. As there is only one trace, counting the number of times a basic block is executed provides exact, absolute execution frequency counts.

The basic idea is similar for code with input-data independent loop counters. However, we need to take nondeterministic control flow branches into account. Instead of absolute execution frequencies, relative ones are recorded to obtain precise flow facts. The framework of abstract execution [4] provides all necessary notions for this analysis.

However, in our setting abstract execution is extremely simplified by eliminating all statements dealing with input-data dependent variables in a preprocessing step.

Generating bounds on relative loop iteration counts works by tracking and merging loop counters. A scope is the set of basic blocks associated with a method or loop. For each scope, loop counter and loop bound variables are introduced for every loop within the scope. The loop bounds are reset at the task entry. Loop counters are reset when a scope is entered. Each time the corresponding loop body is executed, the loop counter is increased. When the scope is left, the loop counter is read, updating the loop bound for the scope/loop pair. Additionally, one counter keeps track of the sum of innermost loops executed in a scope. In this way, it is possible to obtain precise flow facts when there are two or more inner loops with different dependencies on an outer loop counter.

Due to the removal of all input-data dependent assignments, it is not necessary to merge the values of ordinary program variables at any point. Only the loop counters used to extract relative loop bounds need to be merged. This observation significantly reduces the complexity of the analysis.

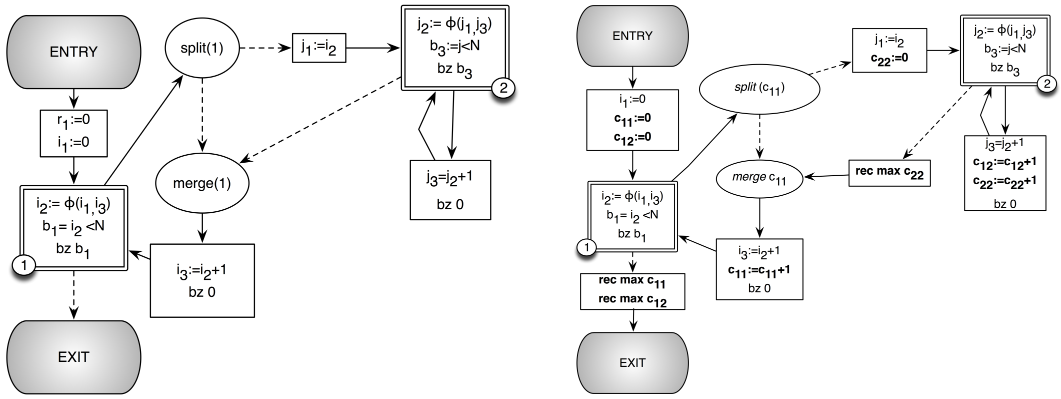
Figure 4a shows the simplified control flow graph of the `dsum` example from Figure 3, with all input-data dependent variables removed. Split points correspond to conditional branches, where the condition variable has been identified as input-data dependent. Note that control flow is split non-deterministically at these nodes, as the condition variable is no longer available.

In Figure 4b, the control flow graph for the abstract execution computing the flow facts for this example is shown. Note that although we used the same programming language for the analyzed input and the generated program carrying out abstract execution, this is not necessary in general. When only a low-level representation of the input is available, it might be desirable to use a higher-level language to generate code for abstract execution. Moreover, for languages with platform-dependent semantics, abstract execution has to faithfully interpret the characteristics of the target platform.

We believe that this form of abstract execution will not have any scalability issues in practice, though experiments with large programs have not been performed yet.

5 Examples and Evaluation

This section discusses important classes of algorithms and tasks with input-data independent loop counters, and investigates the input-data independence of loop counters on a set of selected benchmarks.



(a) Reduced Flow Graph

(b) Abstract Execution Flow Graph

Figure 4 Loop Bound Analysis of the `dsum` Example

Digital Signal Processing

Many algorithms used in digital signal processing do have natural single path implementations, given fixed array, matrix and block sizes. Examples include matrix multiplication, the discrete cosine transform (DCT), the discrete Fast Fourier Transform (FFT) and Finite Impulse Response (FIR) filters. The symbolic loop bound for e.g. FFT is not trivial to find. Given an input-data independent block size, the abstract execution technique from Section 4 determines precise loop bounds, taking non-rectangular loop nests into account. For single-path algorithms, the complexity of calculating a loop bound is comparable to the complexity of simply executing the program.

Search and Sort

Binary search has already served as an example in the introduction (Listing 1). Insertion Sort and iterative Merge Sort are sorting algorithms which use input-data independent loop counters if the size of the array to be sorted is input-data independent. Quick Sort does not, but is unsuitable for hard real-time systems because of its poor worst-case performance.

State Machines

State machines, which perform different actions depending on the value of a state variable, incur a higher overhead when transformed to single-path code. This is because the actions of every state have to be carried out to conform to the single-path requirement. With our new code policy, the loop counters in each action are input-data independent, state machines need not be changed so that the task conforms to the policy.

Data structures with dynamic size

The loops of algorithms operating on data structures with a variable number of elements are usually bounded by a function based on the number of elements, not their maximal capacity. For these algorithms, different variants which are oriented towards the worst-case (size = capacity), need to be used. As it is necessary to distinguish undefined and defined entries, a certain overhead will be unavoidable here. It still remains to be evaluated whether this is an acceptable strategy.

Qualitative Benchmark Evaluation

We manually analyzed the properties of three applications available for the Java Optimized Processor (JOP) [11], and evaluated whether they conform with the policy. We found that most loop bounds do have input-data independent counters, while for those that do not, the dataflow analysis in JOP's WCET tool failed to derive loop bounds as well.

Lift benchmark: The Lift benchmark is the control loop of a simple lift controller. The task performs one out of a few different actions depending on its state and sensor values. All loops in the Lift benchmark have input-data independent counters, with most of them already being identified by the dataflow analysis integrated with JOP's WCET tool. When eliminating all input-data dependent assignments manually, 8 out of 13 methods are removed from the code.

Kfl Benchmark: The Kfl application is the software for a node in a distributed mast control application. All but two loops again had input-data independent counters. The remaining ones need to be annotated manually. The annotations are based on the programmer's knowledge that some global, static variable is always between 0 and 3. This is a non-obvious information, unlikely to be found by an automated analysis. This suggests to rewrite the offending code in order to avoid the annotation.

EjipCmp Benchmark: This benchmark is taken from an implementation of the UDP/IP stack used in a multi-core version of JOP. Some of its loops depend on the number of bytes a message contains. While there is a global limit to this bound, which depends on the size of the array used for storing the message, the message length is not input-data independent. In this case, the easiest way to conform to the policy is to add another exit condition based on the global limit.

6 Discussion

6.1 Source Code versus Machine Code

The ideas presented in this paper work on two different levels: The construction of predictable code applies to the source code level, while the flow fact generation for the compiled code applies to an optimized representation in some lower level language. This may either be low-level C, an internal representation of the compiler, or even machine code. For machine code, the flow graph reconstruction is not easy to automate though.

For the source code level, we need to provide a methodology for building or generating the code, and analysis tools to verify that the code meets the policy. The first goal is met by introducing annotations specifying input data independence via function interfaces and type annotations. To detect input-data dependencies, we perform a dataflow analysis on the source code. It is the responsibility of the programmer to ensure that input-data dependent branches terminate a loop eventually. If this is not the case, the program is considered to be faulty, and abstract execution may not terminate.

6.2 Functional Correctness vs. Timing Analysis

Proving the functional correctness is of course important too, so an interesting question is whether implementations which fulfill the proposed policy are easier or more difficult to prove correct. While we do not know an answer in general, the ability to detect loop bounds by means of static analysis is also beneficial for other program analysis tools. In particular, bounded model checkers, which are used to prove the absence of certain runtime errors (null pointer dereference, out of bound array indices) need to know all loop iteration bounds.

7 Conclusion

In this paper, we have presented a formal definition for a code policy for hard real-time systems. For tasks with input-data independent loop counters, it is guaranteed that all loop bounds can be detected automatically. Furthermore, it is possible to check statically that the policy is fulfilled, and to systematically construct tasks following this policy. We have argued that this policy, which originates from the single-path paradigm, is suitable for real-time systems, and indeed characterizes a large set of analyzable code. Finally, a sketch of a static, efficient implementation of abstract execution to derive all loop bounds has been presented. We have recently started the implementation of the input-data dependency analysis. Future work includes implementation of the simplified abstract execution technique, removing statements dealing with input-data dependent prior to the analysis. Furthermore, we want to investigate suitable algorithms for dynamic data structures, and experiment with the analysis of machine code on ARM targets, which have served as a platform for single-path experiments in the past.

Acknowledgements

The authors would like to thank Wolfgang Puffitsch, Sven Bünte and the anonymous reviewers for their detailed and valuable comments. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design).

References

1 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

2 Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117, New York, NY, USA, 1998. ACM.

3 Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-Aware C Compiler. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 121–126, Seoul/Korea, October 2006.

4 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

5 Jan Gustafsson, Björn Lisper, Raimund Kirner, and Peter Puschner. Code analysis for temporal predictability. *Real-Time Syst.*, 32(3):253–277, 2006.

6 Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

7 Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 29, Washington, DC, USA, 2001. IEEE Computer Society.

8 Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

9 Peter Puschner and Alan Burns. Writing temporally predictable code. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.

10 Peter P. Puschner. Transforming execution-time boundable code into temporally predictable code. In *DIPES '02: Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems*, pages 163–172, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

11 Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

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

WCET Computation of Safety-Critical Avionics Programs: Challenges, Achievements and Perspectives (Invited Talk)

Jean Souyris[1]

1 AIRBUS
316 route de Bayonne, 31060 Toulouse cedex 03, France
jean.souyris@airbus.com

Abstract

Time-critical avionics software products must compute their output in due time. If it is not the case, the safety of the avionics systems to which they belong might be affected. Consequently, the Worst Case Excution Time of the tasks of such programs must be computed safely, i.e., they must not be under-estimated. Since computing the exact WCET of a real-size software product task is not possible (undecidability), "safe WCET" means over-estimated WCET. Here we have an industrial issue in the sense that too over-estimating the WCET leads to a waste of CPU power. Hence, the computation a safe and precise WCET is the big challenge. Solutions to that problem cannot only rely on the technique for computing the WCET. Indeed, both hardware and software must be designed to be as deterministic as possible. For its Flight controls software products, Airbus has always been applying these principles but, since the A380, the use of more complex processors required to move from a technique based on measurements to a new one based on static analysis by Abstract Interpretation. Another kind of avionics applications are the so-called High-performance avionics software products, which are significantly less affected by - rare - delays in the computation of their outputs. In this case, the need for a "safe WCET" is less strong, hence opening the door to different other ways of computing it. In this context, the aim of the talk is to present the challenge of computing WCET in Airbus's industrial context, the achievements in this field and the evocation of some trends and perspectives.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.89

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 89–89
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core*

Christine Rochange[1], Armelle Bonenfant[1], Pascal Sainrat[1], Mike Gerdes[2], Julian Wolf[2], Theo Ungerer[2], Zlatko Petrov[3], and František Mikulu[3]

1 **IRIT - Université Paul Sabatier**
Toulouse, France
`{rochange,bonenfant,sainrat}@irit.fr`

2 **University of Augsburg**
Germany
`{wolf,gerdes,ungerer}@informatik.uni-augsburg.de`

3 **Honeywell International s.r.o.**
Czech Republic
`{Zlatko.Petrov,Frantisek.Mikulu}@Honeywell.com`

Abstract

To meet performance requirements as well as constraints on cost and power consumption, future embedded systems will be designed with multi-core processors. However, the question of timing analysability is raised with these architectures. In the MERASA project, a WCET-aware multi-core processor has been designed with the appropriate system software. They both guarantee that the WCET of tasks running on different cores can be safely analyzed since their possible interactions can be bounded. Nevertheless, computing the WCET of a parallel application is still not straightforward and a high-level preliminary analysis of the communication and synchronization patterns must be performed. In this paper, we report on our experience in evaluating the WCET of a parallel 3D multigrid solver code and we propose lines for further research on this topic.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.90

1 Introduction

The demand for computing power in embedded systems is ever growing as is the demand for new functionalities that will improve safety, comfort, number and quality of services, greenness, etc. Multi-core processors are now being considered as first-rate candidates to achieve high performance with limited chip costs and low power consumption. However, off-the-shelves components do not exhibit enough timing predictability to be used to design hard real-time (HRT) systems since the estimation of worst-case execution times (WCETs) would be infeasible or extremely pessimistic.

The MERASA project focuses on multi-core processors and system-level software for HRT embedded systems. The main issue is to guarantee the analyzability and predictability of WCETs. The proposed MERASA architecture features 2 to 16 cores, each one with simultaneous multithreading (SMT) facilities designed to support one HRT thread and up to three non real-time (NRT) threads. Private instruction and data scratchpad memories favor

[*] **M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is partially funded by the European Community's Seventh Framework Programme under Grant Agreement No. 216415

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 90–100
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

local memory accesses for HRTs and the shared memory hierarchy, including the common bus, features time-predictable arbitration policies. Target workloads are multiprogrammed, mixed-critical tasks, as well as multithreaded, including control and data parallelism. However, computing the WCET of a parallel application is not a straightforward process even when the hardware features timing predictability. As far as we know, this issue has not been addressed so far and the work that is reported in this paper is to be seen as a first attempt from which we will get feedback to guide future research.

Within the MERASA project, a collision avoidance application, developed by Honeywell International and based on the Laplace'equation for a 3D path planning, has been selected as example of an industrial-relevant application to support investigations on timing analyzability and predictability. In this paper, we study the core component of this application, a 3D multigrid solver. The parallelized version of this code splits the 3D domain into 3D compartments and creates one thread to process each compartment. We assume a number of threads equal to or lower than the number of cores in the target MERASA processor so that all the threads can run in parallel, each on one core. While the hardware guarantees isolation between concurrent HRT threads so that their respective WCETs could be analyzed without any difficulty as if they were independent, software-related dependencies due to shared data and synchronizations make the WCET analysis of the whole application challenging. In this paper, we show how we have estimated the WCET of the parallel 3D multigrid solver.

The paper is organized as follows. Section 2 introduces the multi-core processor and system software designed within the MERASA project. The parallel 3D multigrid solver is presented in Section 3. How the WCET of this application can be analyzed is discussed in Section 4 and experimental results are reported in Section 5. Section 6 provides feedback from this study and Section 8 concludes the paper.

2 The MERASA multi-core architecture and system software

The MERASA processor architecture was developed as a timing predictable and WCET-analyzable embedded multi-core architecture. Cores must execute HRT threads in isolation, and the hardware must guarantee a time-bounded access to shared resources by e.g. applying techniques for a real-time capable bus and memory controller [7]. In order to allow mixed application execution of HRT and NRT threads each MERASA core is an in-order SMT-core providing the possibility to run a HRT thread simultaneously in concert with additional NRT threads. The cores of our multi-core processor feature two pipelines, an address and a data pipeline, a first level of hardware scheduling to the thread slots, a real-time aware intra-core arbiter, a data scratchpad (DSP) and a dynamic instruction scratchpad (D-ISP) [5] on core level. The instruction scratchpad is automatically filled with target functions on call/return instructions so that (a) the load time only depends on the function size and can be accounted for when computing the WCET, and (b) every instruction fetch within a function always hit in the D-ISP. The real-time aware intra-core arbiters are connected to the real-time bus for accessing the main memory. The real-time bus features different real-time bus policies inspired from Round-Robin and priority-based schemes [6] for dispatching requests to the memory. Thus, we are able to execute multi-programmed and multithreaded workloads on our multi-core processor. A detailed description of the MERASA multi-core and its implementations as low-level (SystemC) and high-level simulators and also as an FPGA prototype are available at the website of the MERASA project (www.merasa.org).

The MERASA system software [11] represents an abstraction layer between application software and embedded hardware. On top of the MERASA multi-core processor it provides the basic functionalities of a real-time operating system (RTOS) like synchronization functions and memory management facilities to be a fundament for application software. Similar demands as for the processor architecture arise for the RTOS: the challenge is to guarantee an isolation of memory and I/O resource accesses of various HRT threads running on different cores to avoid mutual and possibly unpredictable interferences between HRT threads and therefore also enable WCET analyzability. If common resources are accessed, a time-bounded handling must be guaranteed. The resulting system software executes HRT threads in parallel on different cores of the MERASA multi-core processor. To yield thread isolation, we decided to apply a second level of hardware-based real-time scheduling and devised a thread control block (TCB) interface for the system software. The TCB interface is used to schedule by hardware a fixed number of HRT threads (fixed by the number of cores, one per core) and an arbitrary number of NRT threads into the available hardware thread slots. The commonly used POSIX-compliant mechanisms for thread synchronization, like mutex, conditional and barrier variables are implemented in a time-bounded fashion and a fixed WCET of each function was computed [11]. For the dynamic memory management we chose a flexible two-layered mechanism with memory pre-allocation in the first and the real-time capable TLSF [4] memory allocator in the second layer.

3 The multigrid solver application

3.1 General overview

The software considered in this study is part of a larger application for airbone collision avoidance. Moreover, it is the basic building block that plans a path between the current vehicle position and the current goal position, using the Laplace's equation. The Laplacian algorithm constructs paths through a 3D domain by assigning a potential value of $v(r) = 0$ for r on any boundaries or obstacles, and a potential value of $v(r) = -1$ for r on the goal region. Then Laplace's equation is solved in the interior of the 3D region, guaranteeing no local minima in the interior of the domain, leaving a global minimum of $v(r) = -1$ for r on the goal region, and maxima of $v(r) = 0$ for r on any boundaries or obstacle. A path from any initial point $r(0)$ to the goal is constructed by following the negative gradient of the potential v.

Numerical solutions of Laplace's equation are obtained by partitioning the domain, then iteratively setting the potential at each interior point equal to the average of its nearest neighbors. By varying the grid size (halving the voxel[1] size at each step) from the crudest that still leaves paths between obstacles, to the finest that is required for smooth paths, the iteration converges in a time proportional to the number of voxels in the finest grid. The solution on crude grids is cheap, and is used to initialize the solution on finer grids. This multigrid technique is described in [10].

In the version of the code we considered, the multigrid solver function includes five phases, each of them breaks down into an interpolation step and an iteration step, shown in Table 1. There are no data races in the interpolation code, while those are in the iteration step code.

[1] A voxel is a volume element, representing a value on a regular grid in a 3D space. It is analogous to a pixel in a 2D space.

Table 1 Steps in the multigrid solver algorithm

Interpolation	Iteration

`for (x=0; x<NX; x++)`
`  for (y=0; y<NY; y++)`
`    for (z=0; z<NZ; z++)`
`      v[x][y][z]`
`        = compInterpolate(old_v);`
	`for (i=0; i<NUM_ITE; i++)`
	`  for (x=0; x<NX; x++)`
	`    for (y=0; y<NY; y++)`
	`      for (z=0; z<NZ; z++)`
	`        v[x][y][z]`
	`          = compIterate(v);`

3.2 Parallel version

This code has been parallelized by breaking the 3D domain down into 3D compartments, as illustrated in Figure 1. The main thread creates as many child threads as compartments and each of them performs the computations for one compartment. The main thread orchestrates the phases and steps and enforces the synchronization of the child threads after each step. During interpolation steps, the child threads run independently from each other. However a synchronization at the end is required to ensure that the whole 3D matrix has been processed before starting the iteration step. In the iteration steps, each thread has to synchronize with the threads that produce the data it depends on and with the threads that consume the produced data.

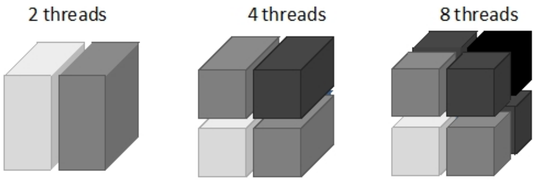


Figure 1 3D domain splitting into compartments

4 WCET analysis of the parallel multigrid solver

4.1 General overview

Analyzing the WCET of a parallel application requires two steps. First, it is necessary to analyze the general structure of the code, and to determine which parts are executed in parallel. The result of this step is a slicing of the code into units and the specification of how units are scheduled with respect to each other. This specification allows deriving the list of code units for which a WCET must be computed and a formula to determine the WCET of the whole application from the WCETs of code units. Second, the synchronizations and communications between threads must be carefully analyzed to be able to compute upper bounds for waiting times due to synchronizations.

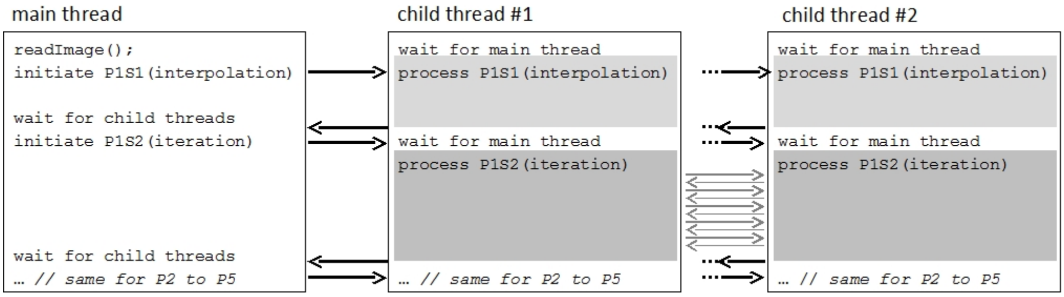
In the following, we will illustrate these two steps considering the multigrid solver application introduced in Section 3.

4.2 Analysis of the application structure

Figure 2 shows the structure of the parallel code, the synchronization points between the main thread and the child threads, as well as the synchronizations between child threads (in this figure, `PiSj` stands for "Phase `i`, step `j`"). From this structure, it is possible to derive a first breakdown of the overall WCET, as shown in Figure 3. This diagram combines the WCETs of code parts, some of them are executed by the main thread and the other ones by the child threads. The question of the synchronizations will be addressed in Section 4.3 but what is suggested here is that the WCET of a code part executed by the main *(resp. a child)* thread does not include the waiting time for other threads: the waiting time is represented by the WCET of child threads *(resp. of the main thread)*. According to the diagram, the WCET can be computed as:

$$WCET_{global} = WCET(main) + \sum_{i=1}^{5} \sum_{j=1}^{2} WCET(P_i S_j)$$

where $WCET(main)$ is computed without accounting for the waiting times.



```
readImage();
initiate P1S1(interpolation)

wait for child threads
initiate P1S2(iteration)



wait for child threads
… // same for P2 to P5
```

```
wait for main thread
process P1S1(interpolation)

wait for main thread
process P1S2(iteration)


… // same for P2 to P5
```

```
wait for main thread
process P1S1(interpolation)

wait for main thread
process P1S2(iteration)


… // same for P2 to P5
```

Figure 2 Structure of the parallel multigrid solver code

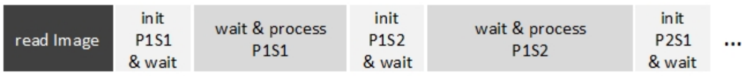


Figure 3 First-level breakdown of the WCET

Now, this application requires a second level of analysis due to the synchronizations between child threads in the iteration steps. As explained in Section 3.2, the sequential algorithm enforces data dependencies between successive iterations of the loop nest since the potential of a voxel is computed from the potential of its neighbors (some of them have been updated before it, other ones will be updated after it). Once the 3D domain is split into compartments, each one has dependencies with the borders of its neighbor compartments. Moreover, the grid is processed several times in a loop.

Fortunately, data sharing patterns are regular and it is possible to compute the WCET of a whole interpolation step as illustrated in Figure 4. The interesting point here is the absence of data races requiring synchronizations between the first compartments and the last ones which allows the first threads starting a new iteration of the outer loop before the last threads have finished the current iteration. This is demonstrated in Figure 4 by the overlapping of i0 threads with i1 threads in the 4-, respectively 8-threaded cases. The examples show that a computation speed-up of at most 2 in the 4-threaded and 4 in the 8-threaded cases can be achieved (not taking the main thread and the synchronization overheads into account).

As a result, if the WCET of each computation part is noted W_i and NUM_ITE is the number of iterations, the WCET of an iteration step can be computed as:

$$WCET(P_iS_2) = \begin{cases} 2.W_i \times \texttt{NUM\_ITE} & \textit{with 2 threads} \\ 3.W_i + 2.W_i \times (\texttt{NUM\_ITE} - 1) & \textit{with 4 threads} \\ 4.W_i + 2.W_i \times (\texttt{NUM\_ITE} - 1) & \textit{with 8 threads} \end{cases}$$

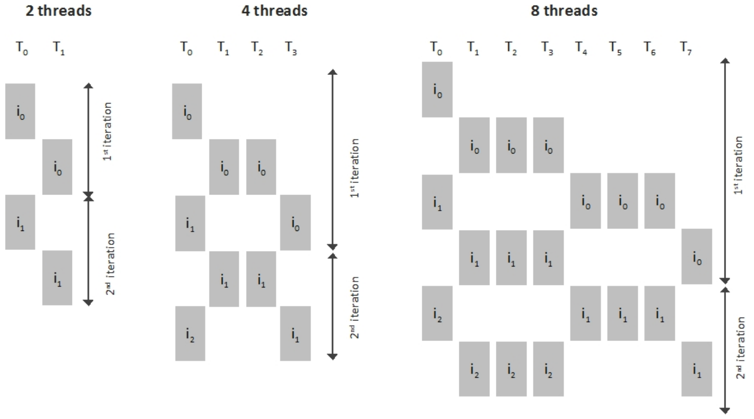


Figure 4 Second-level breakdown of the WCET

4.3 Analysis of the synchronizations

In the application under analysis, the inter-thread synchronizations are implemented using POSIX-compliant mutex and conditional variables. In this Section, we focus on the mutex variable acquisition (`mutex_lock`) but the approach that we propose to analyze the waiting time within this function is also valid for other synchronization primitives.

A simplified version of the `mutex_lock()` function is shown in Figure 5. There are four different cases for a thread trying to acquire the mutex lock. If the mutex lock is free, the thread will acquire the guard lock (1), the mutex lock, and release the guard lock (4). If the mutex lock is hold by another thread, the thread will, after acquiring the guard lock (1), suspend on the mutex lock (2). If the other thread releases the mutex lock, all suspended threads will try to get the mutex lock (3). Now, there are two possibilities: either a thread acquires the mutex lock (4) or it is suspended again (2) if an other thread acquired the mutex lock successfully. The worst-case waiting time for the guard lock is then the maximum WCET of all these four paths and of the similar paths in the other primitives that manipulate the guard lock, multiplied by the number of active threads.

In general, the WCET of a synchronization primitive can be broken down into four terms:

Term		depends on...	
		# threads	application
T_e	Execution time when the synchronization variable is free	no	no
T_{w1}	Overhead execution time when the thread has to wait for the variable	no	no
T_{w2}	Waiting time related to system-level variables	yes	no
T_{w3}	Waiting time related to application-level variables	yes	yes

As far as `mutex_lock()` is concerned, these times can be computed as follows:

- T_e is the WCET computed with flow facts indicating that the loops (the one that implements the wait for the mutex lock, but also the one that stands for the wait for the guard lock in `spinlock_lock()`) have zero iteration, which corresponds to the situation where both the guard lock and the mutex lock are free. This is a constant time.
- T_{w1} is the overhead time of waits (both for the guard lock and the mutex lock), i.e. the time to enter and then leave the loop, considering the lock is released right after the unsuccessful try. In practice, this time is the difference between T_e and the WCET computed considering each loop iterates once (ignoring the waiting time). This time is also constant.
- T_{w2} stands for the waiting times that relate to variables that are manipulated by system-level code only. This is the case for the guard lock. The knowledge of the system software code makes it possible to determine all the possible executions where a thread holds the guard lock. The time T_{w2} then depends on the number of threads but not on the application code, i.e. once the system software has been analyzed, this time is known for any application.
- Finally, T_{w3} includes the waiting times that relate to variables used at the application level. Here, the mutex lock is concerned. How long a thread will have to wait to get the mutex lock depends on the application code and more specifically on possible paths from any call to `mutex_lock()` to any call to `mutex_unlock()`. This time also depends on the number of threads that are likely to use the variable.

```
    int mutex_lock(mutex_t mutex) {
(1)   spinlock_lock(&mutex->guard);
     ...
        while (spinlock_trylock(&mutex->the_lock)) {
         ... // insert thread into queue
(2)      spinlock_unlock_and_set_suspended(&mutex->guard);
(3)      spinlock_lock(&mutex->guard); // on wakeup
       }
       ...
(4)    spinlock_unlock(&mutex->guard);
}
```

Figure 5 Code of the `mutex_lock` primitive

Three kind of synchronizations have to be considered when analyzing the WCET of the parallel multigrid solver. Synchronizations from the main thread to the child threads rely on a shared variable that indicates the next step to execute. It includes a mutex lock (to protect accesses to the shared variable) and a condition to wait on when the variable is not in the expected state. With N child threads, the number of threads that manipulate the lock and the condition is $N + 1$. The paths on which the lock can be hold can be identified in the main thread and child thread codes: the maximum WCET of these paths, multiplied by $N + 1$, makes the T_{w3} term for the lock. Synchronizations from the child threads to the main thread (it must wait that they all have performed the current step before initiating the next one) are similar except for each thread has its own state variable (which is shared only with the main thread). Synchronizations between two child threads (when one produces data used by the other one, on compartment borders) are implemented through state variables related to each child thread. As above, state variables include mutex lock and conditions. Each state variable is shared with the producers and consumers of its owner.

5 Experimental results

In this Section, we provide experimental results obtained considering the following configuration for the MERASA multi-core processor: 2 to 8 cores available for computation threads (an additional core is considered for the main thread), perfect ISP (all the instructions can be fetched from the instruction scratchpad memory), DSP (scratchpad for stack data), round-robin bus, 5-cycle DRAM latency. Due to the round-robin policy and to intra-core arbitration between real-time and non-real-time threads, the *worst-case* latency of an access to the main memory is $5 \cdot n + 12$ for an n-core configuration. In addition, we have considered a single-core configuration to calculate the WCET of a single-threaded version of the application. The application was compiled to the TriCore ISA [13].

WCETs have been computed using OTAWA, a toolset based on static WCET analysis techniques [2]. A specific model of the MERASA architecture has been implemented within this framework. To control the WCET analysis, we have written a script that initiates all the required computations. First, it gets all the WCET terms needed for system-level synchronization functions; then, it requests the WCET analysis of application-level code parts (main function, computation steps); finally, it combines all these times to compute the overall WCET.

Table 2 Estimated WCETs (# cycles)

	1 thread	**2+1 threads**	**4+1 threads**	**8+1 threads**
1 core	53,990,765	-	-	-
3 cores	58,297,375	66,849,369	-	-
5 cores	62,603,985	73,372,109	40,389,428	-
9 cores	71,217,205	86,417,589	47,678,968	28,105,761

Table 2 shows our WCET estimates for several configurations (from 1 to 9 cores) and for several versions of the application (from 1 to 8+1 threads). We considered small grid sizes so that only three phases (instead of five) are executed. As expected, the WCET of a t-threaded version of the application increases with the number of cores: this is due to the round-robin bus policy under which, in the worst case, a given core has to wait for all the other cores to be served before being served itself. Then the worst-case memory latency increases linearly with the number of cores.

On the other hand, the WCET is noticeably improved when the application is parallelized to 4 threads and more. We define the *WCET-speedup* as the WCET of the single-threaded code executed on one core over the WCET of the n-threaded code executed on n cores. On a 9-core architecture, the WCET-speedup is 1.13 with 4 computation threads and 1.9 with 8 computation threads.

Table 3 indicates how the WCET breaks down into "active" execution time (ignoring all waits), waiting time due to synchronizations and waiting time for producing threads (in iteration steps, a computation thread has to wait until all the previous compartments have been updated by other threads before starting to update its own compartment). The provided numbers are for a 9-threaded version of the code and a 9-core configuration, but other combinations exhibit similar breakdowns. It appears that the part of the execution time due to the main thread is small (6%): it mainly includes the time to read the grid.

Moreover the synchronizations have little impact on the overall WCET: they account for less than 2%. This is due to limited interactions between concurrent threads. The largest part of the total WCET (more than 92%) comes from the computation threads. About one half of this time is spent in performing the computations assigned to a thread (three interpolation steps and three iteration steps for the map size selected for this study) while the other half is spent in waiting for producing threads within iteration steps. This is coherent with Figure 4 that shows that the length of loop body in an iteration step should be twice the length of computing one compartment. Predominance of the iteration steps duration in the total WCET explains why the dual-threaded version of the code does not improve the WCET but instead degrades it: since parallelism is not exploited in the iteration steps (both compartments must be processed in sequence), the gain due to parallelized interpolation steps is not sufficient to counterbalance the cost of synchronizations.

Table 3 WCET breakdown (8 computation threads, 9 cores)

WCET for the main thread without waits	6.0%
WCET for a computation thread without waits	45.0%
Time to wait for producers in iteration steps	47.2%
Total waiting time for synchronizations	1.8%

6 Lessons learnt from this case study

While research on WCET computation of sequential programs has received much attention the last fifteen years, resulting in a set of techniques that can handle not too complex software and hardware, the arrival of multi-cores on the embedded systems market raises the need of investigating strategies to analyze the WCET of parallel applications. In this paper, we have reported a study that has been carried out as part of the MERASA project. This first experience in the domain has inspired new lines of research that we will outline in this Section.

The first need for the WCET analysis of a parallel program that appeared during this study is to get an overview of the structure of the application. It is necessary to determine which parts of the code execute in parallel and where the dependencies are. This knowledge is required to build the first-level breakdown of the overall WCET and to schedule the analysis of each piece of code that must be taken into account. Automatically extracting this kind of information from the source or executable code seems infeasible and it is likely that the user/programmer will be asked to provide them. However, this might be a source of errors and it may be desirable to favor well-known parallelization patterns. In addition, an appropriate formalism to express the software parallel architecture would be helpful.

The second need concerns synchronizations: to compute worst-case waiting times, it is necessary to know where synchronizations occur and which threads share synchronization variables. Moreover, the paths on which locks are hold by threads must be carefully identified. Again, specific support should be provided (e.g. in the form of an annotation language) so that the user can specify these information.

In addition, the considered WCET analysis tool must be enhanced to be able to process the specifications of the parallelized code (structure and synchronizations). In particular, we have identified as a must functionality the ability to compute the WCET of a given path (from address a to address b) taking into account its possible contexts of execution. For example, the cache behavior should be preliminary analyzed considering the whole program instead of performing the cache analysis on the path only. So far, the OTAWA toolset does not support global context analysis for path WCET estimation. This is the reason why we have considered a perfect instruction scratchpad in this paper.

Finally, we would like to point out that WCET analysis might not be feasible for any parallel code. As future work, we plan to provide a set of parallelization recommendations that will serve as guidelines to split the computations into parallel threads so that both the code structure and the communication and synchronization patterns match the requirements for timing analyzability.

7 Related work

As we have mentioned it, we are not aware of any paper dealing with the WCET analysis of data-parallel applications and in particular with the analysis of synchronization delays. However, several recent works have studied the impact of interactions between concurrent threads on the WCET.

Some papers focus on the delays introduced by the other threads in the interconnection structure (mainly at the bus level). Solutions reside in predictable arbitration schemes like Round-Robin (considered in the MERASA architecture) [6] or TDMA schemes [1]. A different approach integrates task- and system-level analyses to derive upper bounds for memory latencies [8].

Other contributions concern the analysis of interactions in the level-2 shared cache [12] or solutions to make this analysis more accurate [3][9]. They all focus on spatial conflicts (when two concurrent threads may access the same cache line in a shared cache), not on temporal interferences (i.e. their cache analysis does not take into account the times at which conflicting accesses to the cache will occur but considers instead the worst case).

8 Conclusion

Multi-core processors seem to be key components for the design of future embedded systems because of their ability to provide high performance with low cost and low power consumption. However, existing designs are often not compatible with worst-case execution time analysis since the dynamic sharing of resources (bus, memory hierarchy, etc.) makes it complex if not impossible to determine upper bounds on the delays experienced by a thread running on one of the cores that are due to other threads running on other cores (or on the same core if simultaneous multithreading is supported). The MERASA project fills in this gap by developing a hard-real-time-aware multi-core processor and the appropriate system software. Both levels (hardware and software) have been designed keeping in mind the need of being able to determine upper bounds on pipeline, bus and memory latencies as well as on delays related to dynamic memory management and inter-thread synchronization. As a result, the MERASA multi-core can be used to run mixed multiprogrammed workloads with critical and non critical tasks as well as parallel programs subject to timing constraints.

However, even with WCET-friendly hardware, the WCET analysis of a parallel application is still challenging because it requires having a clear view on the code parallel structure and on the communication and synchronization patterns. We have investigated this field and this paper reports our first experiment in computing the WCET of a parallel 3D multigrid solver code. The process described here is completely guided by the user who must provide insight into how the complete code breaks down into units that can be analyzed separately before their respective WCET estimates are combined to produce the total WCET. As future work, we plan to work on automating the analysis as much as possible.

References

1 Andrei A., Eles P., Peng Z., Rosen J.: Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: 21st International Conference on VLSI Design, 2008.

2 Casse H., Sainrat P.: OTAWA, a Framework for Experimenting WCET Computations. In: 3rd European Congress on Embedded Real-Time Software, 2006.

3 Hardy D., Piquet T., Puaut I.: Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In: IEEE Real-Time Systems Symposium (RTSS), 2009.

4 Masmano M., Ripoll I., Crespo A., Real J.: TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In: 16th Euromicro Conf. on Real-Time Systems (ECRTS), 2004.

5 Metzlaff S., Uhrig S., Mische J., Ungerer T.: Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors. In: 9th workshop on MEmory performance (MEDEA), 2008.

6 Paolieri M., Quiñones E., Cazorla F., Valero M.: Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In: 36th Int'l Symp. on Computer Architecture (ISCA), 2009.

7 Paolieri M., Quiñones E., Cazorla F., Valero M.: An Analyzable Memory Controller for Hard Real-Time CMPs. In: IEEE Embedded Systems Letters, 1(4), 2009.

8 Stachulat J., Schliecker S., Ivers M., Ernst R.: Analysis of Memory Latencies in Multi-Processor Systems. In: 5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.

9 Suhendra V., Mitra T.: Exploring Locking and Partitioning for Predictable Shared Caches on Multi-cores. In: 45th Conf. on Design Automation (DAC), 2008.

10 Valavinis K.P., Herbert T., Kollura R.,Tsourveloudis N.: Mobile Robot Navigation in 2-D Dynamic Environments Using an Electrostatic Potential Field. In: IEEE Trans. on Systems, Man, and Cybernetics-PART A: Systems and Humans, 30(2), 2000.

11 Wolf J., Gerdes M., Kluge F., Uhrig S., Mische J., Metzlaff S., Rochange C., Casse H., Sainrat P., Ungerer T.: RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In: 13th IEEE Int'l Symp. on Object/component/service-oriented Real-time Distributed Computing (ISORC), 2010.

12 Yan J., Zhang W.: WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2008.

13 Infineon Technologies AG, TriCore 1 User's Manual (v1.3.8), Jan., 2008.

Towards WCET Analysis of Multicore Architectures Using UPPAAL*

Andreas Gustavsson[1], Andreas Ermedahl[1], Björn Lisper[1], and Paul Pettersson[1]

1 **School of Innovation, Design and Engineering, Mälardalen University**
Box 883, S-721 23 Västerås, Sweden.
`{andreas.sg.gustavsson,andreas.ermedahl,bjorn.lisper,paul.pettersson}@mdh.se`

Abstract

To take full advantage of the increasingly used shared-memory multicore architectures, software algorithms will need to be parallelized over multiple threads. This means that threads will have to share resources (e.g. some level of cache) and communicate and synchronize with each other. There already exist software libraries (e.g. OpenMP) used to explicitly parallelize available sequential C/C++ and Fortran code, which means that parallel code could be easily obtained.

To be able to use parallel software running on multicore architectures in embedded systems with hard real-time constraints, new WCET (Worst-Case Execution Time) analysis methods and tools must be developed. This paper investigates a method based on model-checking a system of timed automata using the UPPAAL tool box. It is found that it is possible to perform WCET analysis on (small) parallel systems using UPPAAL. We also show how to model thread synchronization using spinlock-like primitives.

1998 ACM Subject Classification C.0, C.1.2, C.3, D.2.4

Keywords and phrases WCET, Multicore, Parallel, Thread Synchronization, Model-Checking, UPPAAL

Digital Object Identifier 10.4230/OASIcs.WCET.2010.101

1 Introduction

The execution of hard real-time systems must be predictable in order to ensure a certain system behavior. In particular, the WCETs (Worst-Case Execution Times) of the hard real-time tasks are assumed to be known and given as input to different real-time system scheduling algorithms [4, 10, 17]. The WCET of a task is dependent both on the properties of the software which is executed as well as the underlying hardware. Today, there are algorithms and tools which strive to derive a safe and tight bound on the WCET of a task, using the task code and a model of the (single-core) target hardware. Some examples of such tools are aiT [9, 27], SWEET [8, 27] and RapiTime [23, 27].

Over the past years, there has been (and there will probably continue to be) a rapid increase in the usage of multicore architectures in embedded real-time systems. These architectures have several independent processing units (cores) on each chip. The cores typically share some resources (e.g. some level of on-chip cache) which introduces dependencies among the cores. Thus the cores could experience delays due to simultaneous access to these shared resources; e.g., if the L1 caches are non-shared and the L2 cache is shared, two

* This work was funded by VR through the project 2008-4650 Worst-Case Execution Time Analysis of Parallel Systems.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 101–112
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

simultaneous misses in the L1 caches will cause one of the cores to delay while the other core is granted access to the L2 cache. If there are one or more levels of core-individual (non-shared) caches, some memory coherence and consistency model will probably be implemented. This means that a line in the local cache of one core may be invalidated by another core's cache, thus introducing a cache miss if the line is again referenced [1].

To take full advantage of these new kinds of architectures, algorithms will need to be parallelized over multiple threads. This means that the threads will have to share resources and communicate and synchronize with each other. There already exist software libraries used to explicitly parallelize sequential code – one example available for C/C++ and Fortran code running on shared-memory machines is OpenMP [20]. The conclusion is that parallel software running on parallel hardware is already available today and will probably be the standard way of computing in the future.

This means that new algorithms, methods and tools for WCET analysis are needed to guarantee the schedulability and predictability of this new kind of systems, where a task could consist of several cooperating threads running in parallel on individual cores. This paper presents a method for WCET analysis of parallel (or sequential) code executing on shared-memory multicore (or single-core) architectures, using verification techniques (model-checking) on a system of timed automata. The paper shows that it is possible to model and analyze the impact on the WCET from having a memory hierarchy consisting of core-individual L1 instruction and data caches, and a shared L2 cache. It also shows how a mutual exclusion software primitive similar to a spinlock could be modeled.

The organization of the rest of this paper is as follows. Section 2 presents some related research performed on analysis of multicore architectures. Section 3 contains an introduction to timed automata and the modeling tool box UPPAAL [5]. Section 4 describes the models and verification queries used to calculate the WCET estimate of an example program. Section 5 contains a discussion of the proposed method. It also suggests several aspects of the method that should be further investigated.

2 Related Work

The idea of using model-checking to perform WCET analysis has been investigated and shown to be adequate for analyzing parts of a single-core system in [14] and [19]. However, to the best of our knowledge, no prior research has been conducted regarding multicores with complete (and non-perfect) memory hierarchies. This aspect is investigated in this paper.

In [18] and [28], model-checking is used to perform WCET analysis. Both papers are closely related to the work presented herein, but mainly propose methods to reduce the state space by altering the program model without affecting the true WCET of the program. Our approach is more focused on analyzing the impact on the WCET from allowing synchronizing tasks. In [28], a perfect data cache is assumed (i.e., all accesses are assumed to be hits), which is generally not the case. In contrast, this paper assumes a complete and non-perfect memory hierarchy. In [29] and [30], static analyses of shared L2 instruction caches are presented. Also in these papers, perfect L1 data caches are assumed.

Other than this, to the best of our knowledge, there mainly exist different techniques used to increase the predictability and analyzability (e.g. to tighten the WCET estimate) of multicore systems. In an extension to the method presented in [29], memory bits for each instruction are used to determine whether the instruction should be cached or not [12] – e.g., to avoid pollution of the shared cache, "Static Single Usage" [12] instructions should not be cached. This generates the possibility to determine a tighter WCET estimate.

In [21], arbiters (hardware circuits) are added to a shared-memory multicore processor to synchronize the memory accesses in order to increase the timing-predictability of the system. The result is a multicore architecture that can be analyzed with existing single-core WCET analysis tools.

GAMC [22] is an SDRAM controller which upper bounds the delay a core can suffer from memory-interferences from other cores. This is an important aspect since the largest memory access latency will occur when accessing the main memory. The result is a tight WCET estimate which only differs at most 13% from the largest measured execution time. Similarly, in [4] and [24], TDMA-based memory bus access policies are introduced to make all memory access latencies predictable, regarding the WCET.

3 Timed Automata & UPPAAL

Timed automata[1] [3] can be used to model real-time systems. An automaton can be viewed as a state machine with locations and edges [15]. A state represents certain values of the variables in the system and which location of an automaton is active, while the edges represent the possible transitions from one state to another [15]. (Continuous) time is expressed as a set of real-valued variables modeling clocks. In UPPAAL, all clocks are initialized to zero and then increase with the same rate [7].

A transition is enabled (i.e., it is possible to perform the particular transition from one state to another) if its accompanying guard is satisfied. A guard can simply be viewed as a boolean expression (which can include variables and clocks) which enables or disables the edge. The guard cannot force the transition to be taken however [7]. When a transition is taken, actions can be performed (e.g., variables can be updated and clocks can be reset to zero).

UPPAAL[2] [5, 16, 26] is a tool used to model, simulate and verify *networks* of timed automata [5, 7, 15]. The automata can synchronize via channels on transitions. Only two automata are allowed to synchronize via a given regular channel at a time. Channels can also be declared as being broadcast, which means that one issuing automaton can synchronize with an arbitrary number (including zero) of waiting automata. Another possibility is to declare a channel as being urgent, which means that when a transition is enabled, it will be performed without allowing any time to pass.

Locations in an UPPAAL timed automaton can have special properties as well; urgent or committed. When a location with one of these properties is active, time is not allowed to pass. The difference between urgent and committed locations is that if there are committed locations active, an outgoing transition from one such location must be taken in the next step – if such a transition does not exist or is not enabled, the system will deadlock. A location in the automaton can have an invariant associated with it. An invariant is a clock constraint which limits the amount of time for which the location is allowed to be active.

Some other features of UPPAAL are a C-like programming interface to ease the modeling task, and meta-variables [5]. If the only difference between two states is the values of variables declared as meta, then the states are considered to be the same. This is useful for reducing the size of the state space while verifying properties of the system. Care should be taken to avoid using meta-variables in a way that could eliminate states from the analysis that actually

[1] The formal syntax and semantics of timed automata can be found in e.g. [2] and [15].

[2] An introduction to UPPAAL and the formal semantics of networks of timed automata are given in [5] and [15] respectively.

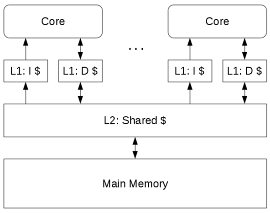


Figure 1 The modeled architecture.

Property	L1-I	L1-D	L2
Lines	4	4	8
Words/Line	2	2	4
Sets	2	4	2
Latency	1	1	10
Replacement Policy	LRU	LRU	LRU

Table 1 Cache Properties.

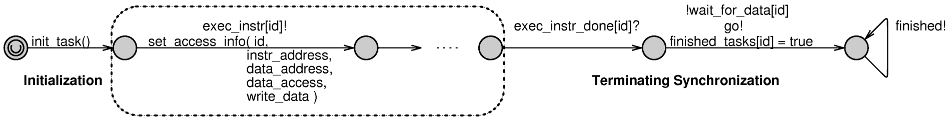


Figure 2 Model of the task interface.

should be taken into account, though. Verification of system properties (requirements) is performed by formulating queries used by the UPPAAL verifier. The query language is described in e.g. [5] or in the help session accompanying the UPPAAL binaries [26].

4 WCET Analysis Using UPPAAL

To model a fictitious shared-memory multicore architecture, a network of timed automata is created in UPPAAL[3]. The architecture is assumed to have the properties depicted in Figure 1; i.e., core-individual L1 instruction and data caches, and a shared L2 cache. In the figure, the arrows between the cores and the caches show the possible flow of memory contents (i.e., instructions and data). The core is assumed to be very simple, only incorporating a pipeline similar to a basic five-stage, in-order RISC-pipeline. The caches are assumed to have the properties found in Table 1.

The resulting models are presented in Figure 3. For a multicore architecture with n cores, there will be n sets of the models in Figures 3a–3c (i.e., one set per core) but only 1 set of the models in Figures 3d–3g[4]. For the current approach, no value analysis is used. Therefore, in the below given models, no actual memory contents is ever transferred or kept track of in the memory hierarchy. The only thing considered is what memory locations (addresses) are referenced by the program. A limitation of this approach is that dynamic memory references cannot be easily modeled.

4.1 The Program Model Interface

The interface for modeling a thread is shown in Figure 2. The “Initialization” part is optional and the `init_task()` function could simply be empty. The “Terminating Synchronization” part ensures that no time is missed by the WCET analysis. If the pipeline should be emptied at the end, a delay should be inserted to account for this in this part of the model.

[3] UPPAAL version 4.0.10 (rev. 4417) has been used in this paper.

[4] With one exception regarding the Lock handler automaton – there is one Lock handler per lock, i.e., per critical section.

The middle (framed) part depicts the instruction execution interface. The instructions are assumed to be assembly instructions and are executed one by one. An instruction is executed by synchronizing with the core automaton via the `exec_instr[id]` urgent channel and setting information about the access via the function call `set_access_info()`. The arguments should be interpreted as: `id` – the core on which the instruction should be executed; `instr_address` – the memory address where the instruction is stored; `data_address` – the address in memory on which the data accessed by the instruction is stored (only used for instructions such as LOAD and STORE etc.); `data_access` – a boolean telling whether the instruction is a data accessing instruction (e.g., a LOAD or STORE etc.); `write_data` – a boolean distinguishing between read and write instructions (i.e., whether the instruction is a LOAD or STORE etc.).

Other types of instructions, such as branch instructions and instructions not referencing memory locations, should be accounted for by adapting the structure of the automata modeling the program. Thus, the structure of the program should be represented by the structure of the automata. This representation could be automatically generated using flow facts generated by a static analysis tool, such as SWEET [8]. The translation would be close to 1:1 of the instruction-level CFG (Control Flow Graph) [18]. To account for hazards, extra stalls can be inserted into the pipeline by setting the `stalls[id]` variable to the desired value before executing the instruction.

To account for the possible memory locations that a given instruction could reference, a value analysis could be used [27]; and to account for the possible values of different variables affecting the execution pattern of the program, a control flow analysis could be used [27]. The structure of the automata modeling the program could then be adapted accordingly (e.g. by adding one transition for each possible memory reference or variable value). This means that UPPAAL will automatically account for the (global) worst-case memory reference or variable value. This approach could also avoid unwanted effects from timing anomalies since UPPAAL searches the entire state space when finding the WCET estimate.

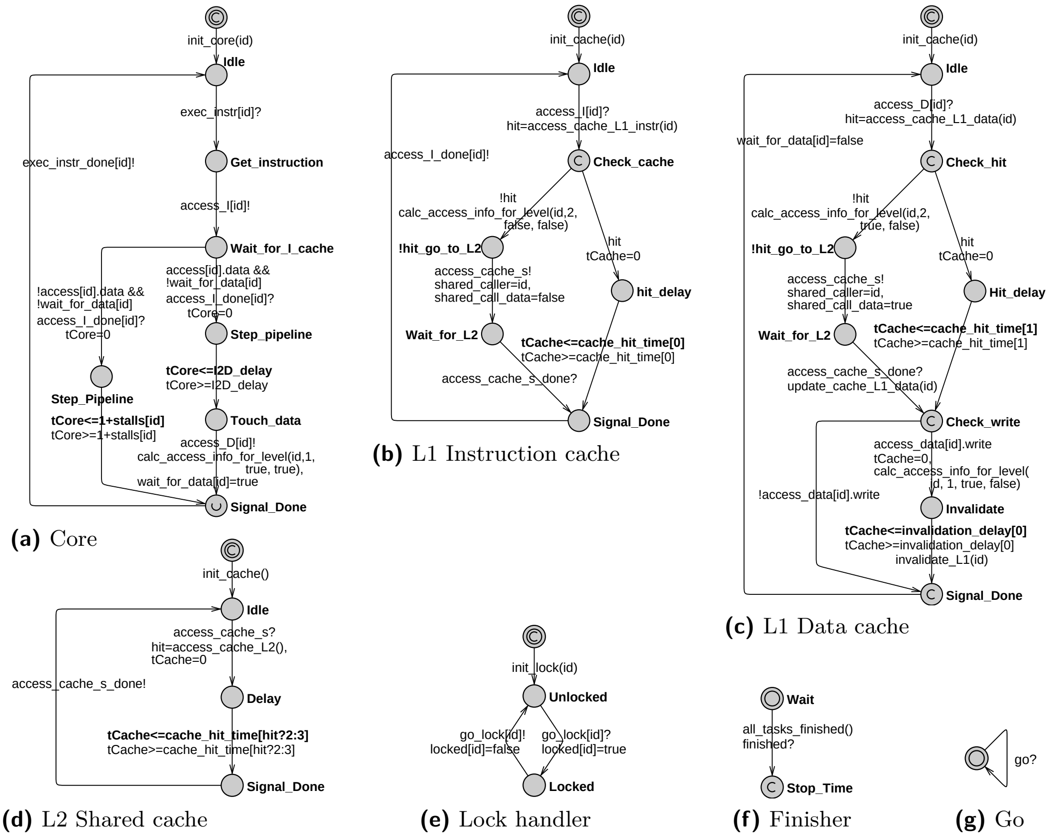
4.2 The Model of the Core

The model of the core is depicted in Figure 3a. This automaton represents the timing model of the core (the pipeline etc.) and is the automaton with which the program-automaton synchronizes to execute instructions. When an instruction should be executed, the core accesses the memory hierarchy to fetch it and then steps the pipeline. If the instruction accesses data, the pipeline is stepped (stalls are inserted) until the memory access stage is reached, then the data is accessed. This leads to an over-approximation of the execution time. However, to avoid further over-approximation (which could be much larger), another instruction can be fetched while the data is accessed.

The `exec_instr_done[id]` channels are declared as broadcast so that the program-automata do not have to synchronize via these channels before a request to execute a new instruction can be issued. This is to minimize the number of locations in the program-automaton (to make the interface as clean as possible and to minimize the state space).

4.3 The Models of the Caches

The models of the L1 instruction and data caches are depicted in Figure 3b and 3c respectively. The main difference between these cache models is that a data cache has the ability to invalidate a line in the other data caches. Otherwise the models are quite straightforward.



(a) Core

(b) L1 Instruction cache

(c) L1 Data cache

(d) L2 Shared cache

(e) Lock handler

(f) Finisher

(g) Go

Figure 3 Timing model of the considered multicore architecture.

All the cache content handling is performed by the `access_cache_L1_{instr,data}()`, `update_cache_L1_data()` and `invalidate_L1()` functions.

If the accessed data is not available in the L1 cache, it is fetched from the L2 shared cache, which is depicted in Figure 3d. This model is even more straightforward – all the cache content handling is performed by the `access_cache_L2()` function. If the accessed data is not located in the L2 cache, it is fetched from the main memory (which is assumed to always hit).

All the caches in the system can be individually defined, regarding set-associativity, cache size, block size and replacement policy (the used cache properties can be found in Table 1).

4.4 The Auxiliary Automata

These automata, depicted in Figures 3e–3g, are implementation specific. The Lock handler-automaton can be (and is) used to implement spinlocks. The Finisher-automaton is used to stop the time and deadlock the system when all tasks have finished executing. And finally, the Go-automaton is very versatile. It simply waits to synchronize via an urgent channel (thus not allowing any time to pass when the transition is enabled). This can be viewed as a trick to achieve the desired system behavior (e.g. to achieve system progress).

4.5 WCET Analysis by Verification

Given the above described network of timed automata, UPPAAL can verify if different properties hold for the system. The verification property that is used to find the WCET estimate looks like[5]: `A[] t <= x`. This property should be interpreted as: "For every possible state, the value of the clock `t` is always less than or equal to `x`". The WCET analysis is easily performed by running the model-checker (verifier) in a binary search style by altering the value of `x` until the WCET estimate is found[6].

In order for this approach to work, some other properties of the system must also be verified; otherwise there might exist some amount of time that is not accounted for when calculating the WCET estimate, or the overall system behavior could be incorrect. It must be verified that: whenever the system is in a deadlock state, the Finisher automaton is in its Stop_Time location; the system will always reach a state where the Finisher automaton is in its Stop_Time location; when the Finisher automaton is in its Stop_Time location, all other automata modeling the hardware are in their Idle locations, and all automata modeling the program have finished; and mutual exclusion is guaranteed on critical sections. By using similar verification properties to the one above, UPPAAL can check these properties automatically[7].

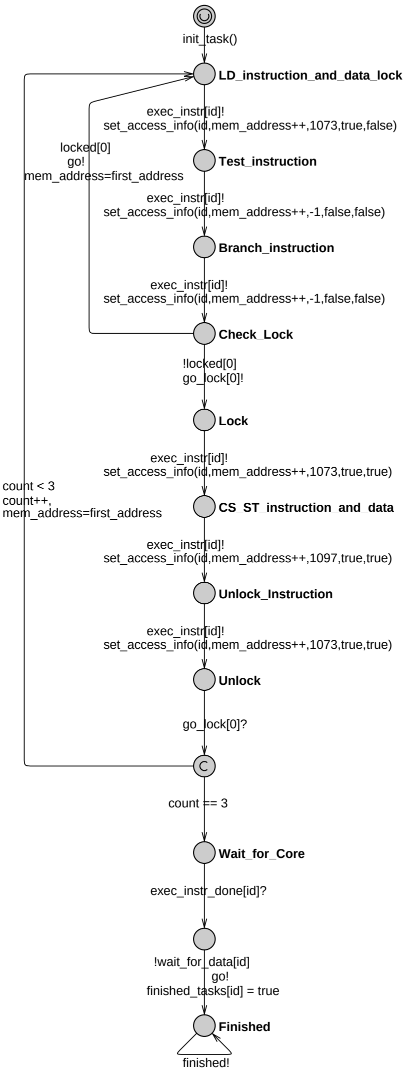

init_task()
LD_instruction_and_data_lock
exec_instr[id]!
set_access_info(id,mem_address++,1073,true,false)
locked[0]
go!
mem_address=first_address
Test_instruction
exec_instr[id]!
set_access_info(id,mem_address++,-1,false,false)
Branch_instruction
exec_instr[id]!
set_access_info(id,mem_address++,-1,false,false)
Check_Lock
!locked[0]
go_lock[0]!
Lock
exec_instr[id]!
set_access_info(id,mem_address++,1073,true,true)
count < 3
count++,
mem_address=first_address
CS_ST_instruction_and_data
exec_instr[id]!
set_access_info(id,mem_address++,1097,true,true)
Unlock_Instruction
exec_instr[id]!
set_access_info(id,mem_address++,1073,true,true)
Unlock
go_lock[0]?
C
count == 3
Wait_for_Core
exec_instr_done[id]?
!wait_for_data[id]
go!
finished_tasks[id] = true
Finished
finished!

Figure 4 Model of a program with spinlock-like synchronization.

4.6 Experimental Evaluation

An example model of a program (using the interface given in Figure 2) is given in Figure 4. The task of the modeled program is very simple; it just acquires a spinlock-like lock and then writes to a shared variable before releasing the lock, and it executes this procedure three times before finishing its execution.

The same task is run on two cores (both tasks are released at the same time) and the result of the analysis is a WCET estimate equal to 636 clock cycles (the other properties mentioned above are also satisfied); using the specific values of the cache sizes (Table 1) and latencies etc. The main memory is assumed to have a latency of 80 clock cycles. Each step in the binary search approach is performed within 1 second and the total number of steps is 11 (this is dependent on the initial values of `x` in the verification property from section 4.5, however).

[5] The UPPAAL verifier syntax can be found in [5] or in the online help session accompanying the UPPAAL binaries [26].

[6] Similar approaches to WCET analysis using model-checking are described in [18], [19] and [28].

[7] To guarantee a safe verification, the UPPAAL option "Extrapolation" should be set to "None".

An initial investigation of some potential problems regarding the scalability of the model-checking approach has been conducted. By increasing the number of cores to four and running one instance of the same example program as above on each core, we get a large slowdown in the analysis time. Another investigation, where the release time of the second task is made general in the interval $[0, 1000]$, has also been performed. The same result, a large slowdown in the analysis time, was observed. Increasing the sizes of the (meta-declared) caches to 2048 lines for the L1 caches and 8192 lines for the L2 cache, does not seem to have an equally large impact on the analysis time though. The memory usage increases drastically, however. The required times for performing one binary search step are summarized in the table below (a dual-core processor, running at 2.66GHz, with 4GB of RAM was used). The "2 Cores" column represents the original experiment and is the base for comparison. The total time is an approximation of the total time needed to perform the analysis, assuming 11 iterations, and that the binary search strategy for invoking the UPPAAL verifier is handled by a script.

	2 Cores	4 Cores	Release Time	$ Sizes
Time	<1s	>3h (aborted)	44s	14s
Total Time	11s	>33h	500s	150s

A consequence of these results is that the complexity of the models and the size of the analyzed program (and thus the achievable tightness of the WCET estimate) have to be balanced to avoid making the state space explode. The case with 4 cores was aborted after approximately 3 hours when the virtual memory demands exceeded the available amount of RAM (4GB).

5 Discussion & Future Work

Modeling systems is very easy using UPPAAL, which also offers a useful interface for performing model-checking. This paper has shown that WCET analysis of parallel code and hardware can be performed using the model-checking techniques available in e.g. UPPAAL. There are some limitations imposed by using UPPAAL to perform the WCET analysis, however. The C-style interface is a bit limited regarding function calls; e.g., an array-argument must have a known size – this limits the level to which the code can be written in a generical way. However, the UPPAAL C-functions are meant to be very simple and small and the C-style interface offered by UPPAAL is in general very rich, so the pros very much outweighs the cons.

Another drawback is the binary search strategy that has to be used for finding the WCET estimate. This could lead to unnecessarily large overheads in the analysis. One way to avoid the binary search approach is to use the new `sup`[8]-operator, implemented in (and described in the help session accompanying) the development version (4.1) of UPPAAL [26]. The `sup`-operator finds the maximum value of an expression evaluating to either an integer or a clock. To find the WCET estimate using the `sup`-operator, the following property could simply be verified: `sup: t`. This property should be interpreted as: "Find the maximum value of the clock `t`". This approach works for the proposed system model since the system is deadlocked and the time is stopped when all tasks have finished executing. The reason to why this approach is not used in this paper is because of the development (unstable) state of the UPPAAL-version (4.1) in which the `sup`-operator is implemented.

[8] `sup` is an abbreviation of suprema.

However, an initial investigation using the sup-operator has been performed on the system described in section 4.6. By verifying the property sup: t, it is found that the WCET estimate is 636 clock cycles (the same result as achieved by using the binary search approach). The total time needed to verify the property is in the order of 1 second – this is superior to the binary search approach where approximately 1 second (plus the overhead needed to adjust the parameters) is needed for each binary search step.

An investigation of the sup-operator's impact on the scalability has also been conducted for the same system setups that were described in section 4.6. The result is presented in the table below.

	2 Cores	4 Cores	Release Time	$ Sizes
Time	1s	>3h (aborted)	42s	14s
Total Time	1s	>3h	42s	14s

As for the binary search approach, the case with 4 cores was aborted after approximately 3 hours when the virtual memory demands exceeded the available amount of RAM (4GB). As can be seen, the total time needed to perform the entire analysis using the sup-operator is quite comparable to the time needed to perform one binary search step (excluding any parameter adjustment overhead). This makes the sup-operator a very promising feature of UPPAAL; since the entire analysis can be performed automatically (in one step) and the implied overhead, if any, is negligible.

Further investigations should be performed, regarding how well this method (model-checking) scales with the size of the modeled program and the complexity of the hardware model. It would also be worth investigating the impact on the size of the state space (and thus the analysis time) by transferring more of the cache handling functionality from the cache automata to the cache handling C-functions, and vice versa. On one extreme, all the cache handling could be done by the C-functions, while the automaton only is used to perform the cache access delay.

Another way of (hopefully) increasing the scalability of the method is to extend the use of scalars. When scalars are used, UPPAAL can apply symmetry reduction on the model [13], which can lead to a dramatic decrease in the size of the state space. Symmetry reduction eliminates redundant paths in the model. Considering the models presented in section 4, there are lots of redundant paths. The same program is executed on several homogenous cores with a homogenous memory hierarchy. This means that the same execution pattern exists several times in the state space, the only difference is which program (and core and caches) it concerns. As a simple example, either program 0 is considered to start before program 1, or vice versa – only one of the possibilities needs to be considered since the models are equal; this is what symmetry reduction tries to achieve. Scalars and symmetry reduction are also described in more detail in the UPPAAL help session[9].

The granularity of the proposed interface in this paper is on the instruction level. This increases the size of the state space compared to using a basic block granularity. One way of reducing the size of the state space, and keep the instruction level granularity (when considering non-preemptive tasks at least), could be to merge instructions on the same cache line that do not access data and add some additional delay in the program model to represent the merging. This would be possible since the lines in the (non-shared) instruction cache never are invalidated by another cache; if one instruction is available, all other instructions in

[9] The UPPAAL help session accompanies the UPPAAL binaries, available at [26]. It is also available at http://www.uppaal.org/help.php?file=WebHelp (for the official release of UPPAAL).

the same line are also available. This approach can be viewed upon as a manually performed partial order reduction [6, 11].

The static WCET analysis tool SWEET[10] is already capable of generating models in the UPPAAL syntax on a special format [25]. Performing minor changes to this generation could adapt SWEET to also being able to create models on the format specified by this paper. This means that benchmarks could be easily translated and analyzed together with the hardware models presented herein.

Other suggestions for future work are to implement a more detailed timing model to avoid over-approximating the WCET, to implement a model of a real-world multicore architecture, such as e.g. the ARM Cortex, and to investigate the possibilities of implementing models of more synchronization primitives, e.g. mutexes and condition variables.

A final and very important conclusion is that WCET analysis of the inter-thread communication and interferences on shared resources can be made quite simple using the suggested model-checking method, compared to static analysis (see e.g. [29]). However, it will probably be quite difficult to make the model-checking method scale well.

References

1 Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical report, Rice University and Western Research Laboratory, 1995.

2 Rajeev Alur. Timed Automata. In *Computer Aided Verification*, volume 1633/1999. Springer Berlin / Heidelberg, January 1999.

3 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, April 1994.

4 Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosén. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. *International Conference on VLSI Design*, 21:103–110, 2008.

5 Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, volume 3185/2004, pages 200–236. Springer Berlin / Heidelberg, December 2004.

6 Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In *CONCUR'98 Concurrency Theory*, volume 1466/1998, pages 41–49. Springer Berlin / Heidelberg, February 1998.

7 Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098/2004, pages 87–124. Springer Berlin / Heidelberg, July 2004.

8 Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Sweden, June 2003.

9 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)*, pages 153–163, March 2007.

10 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 387–397, December 2009.

11 John Håkansson and Paul Pettersson. Partial Order Reduction for Verification of Real-Time Components. In *Formal Modeling and Analysis of Timed Systems*, volume 4763/2007, pages 211–226. Springer Berlin / Heidelberg, September 2007.

[10] SWEET uses basic blocks by default [8], but does also have the capability of using an instruction level granularity. This makes interaction possible.

12 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 68–77, 2009.

13 Martijn Hendriks, Gerd Behrmann, Kim G. Larsen, Peter Niebert, and Frits Vaandrager. Adding Symmetry Reduction to UPPAAL. In *Formal Modeling and Analysis of Timed Systems*, volume 2791/2004, pages 46–59. Springer Berlin / Heidelberg, May 2004.

14 Benedikt Huber and Martin Schoeberl. Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis. In *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, 2009.

15 Joost-Pieter Katoen. Concepts, Algorithms, and Tools for Model Checking. In *Lecture Notes of the Course "Mechanised Validation of Parallel Systems" (course number 10359)* Semester 1998/1999, at Friedrich-Alexander Universität Erlangen-Nürnberg.

16 Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, December 1997.

17 Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

18 Mingson Lv, Nan Guan, Wang yi, Qingxu Deng, and Ge Yu. Efficient Instruction Cache Analysis with Model Checking. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session*, pages 33–36, April 2010.

19 Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In *Computer Aided Verification*, volume 3114/2004, pages 298–301. Springer Berlin / Heidelberg, July 2004.

20 OpenMP. OpenMP Application Program Interface, Version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

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

22 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. GAMC: A Generic Analyzable Memory Controller for Hard Real-Time Multicore Processors. Technical report, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, May 2009.

23 Rapitime. Rapitime white paper, 2009. `www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf`.

24 Jakob Rosén, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS'07)*, pages 49–60, 2007.

25 Daniel Sundmark. *Structural System-Level Testing of Embedded Real-Time Systems*. PhD thesis, Mälardalen University, Department of Computer Science and Electronics, Sweden, 2008.

26 UPPAAL. UPPAAL Website, 2010. `http://uppaal.org`.

27 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

28 Lan Wu and Wei Zhang. Bounding Worst-Case Execution Time for Multicore Processors through Model Checking. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session*, pages 17–20, April 2010.

29 Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *Proc. 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 80–89, June 2008.

30 Jun Yan and Wei Zhang. Accurately Estimating Worst-Case Execution Time for Multi-Core Processors with Shared Direct-Mapped Instruction Caches. In *Proc. 15th International Conference on Real-Time Computing Systems and Applications (RTCSA'09)*, pages 455–463, August 2009.

METAMOC: Modular Execution Time Analysis using Model Checking

Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René R. Hansen, and Kim G. Larsen

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark
{andrease, mchro, mt, rrh, kgl}@cs.aau.dk

Abstract

Safe and tight worst-case execution times (WCETs) are important when scheduling hard real-time systems. This paper presents METAMOC, a modular method, based on model checking and static analysis, that determines safe and tight WCETs for programs running on platforms featuring caching and pipelining. The method works by constructing a UPPAAL model of the program being analysed and annotating the model with information from an inter-procedural value analysis. The program model is then combined with a model of the hardware platform and model checked for the WCET. Through support for the platforms ARM7, ARM9 and ATMEL AVR 8-bit, the modularity and retargetability of the method are demonstrated, as only the pipeline needs to be remodelled. Hardware modelling is performed in a state-of-the-art graphical modelling environment. Experiments on the Mälardalen WCET benchmark programs show that taking caching into account yields much tighter WCETs than without modelling caches, and that METAMOC is a sufficiently fast and versatile approach for WCET analysis.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.113

1 Introduction

Embedded software is virtually ubiquitous these days. It is used to control the proper functioning of technical devices we routinely use and rely on in our daily life. Often embedded software is applied in safety-critical systems—e.g. the braking system of a car or the steering gear of an airplane. Many of these safety-critical systems are also time-critical, meaning that the calculations performed by the tasks of an embedded system need not only be correct but must be carried out in a timely fashion. Worst-case execution time (WCET) analysis is concerned with providing guarantees for proper timing behaviour of system tasks by computing bounds for their execution time on given processors.

In order to allow for reliable and efficient scheduling of tasks, the scheduling algorithms need safe and tight WCETs. Two different classes of methods are predominant (see also [11]): *measurement-based methods*, where statistical information on WCETs is obtained by executing tasks on the given processor or simulator for a sample collection of input, and *static methods*, where static analysis (typically abstract interpretation and integer linear programming [10]) of the task, taking the specific hardware platform into account, allow the derivation of *safe* upper bounds on the execution time. The method presented in this paper, Modular Execution Time Analysis using Model Checking (METAMOC)[1], is a static method utilising model checking to provide safe WCET estimates. Figure 1 provides an overview of the prototype implementation of METAMOC.

[1] http://metamoc.dk

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 113–123
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

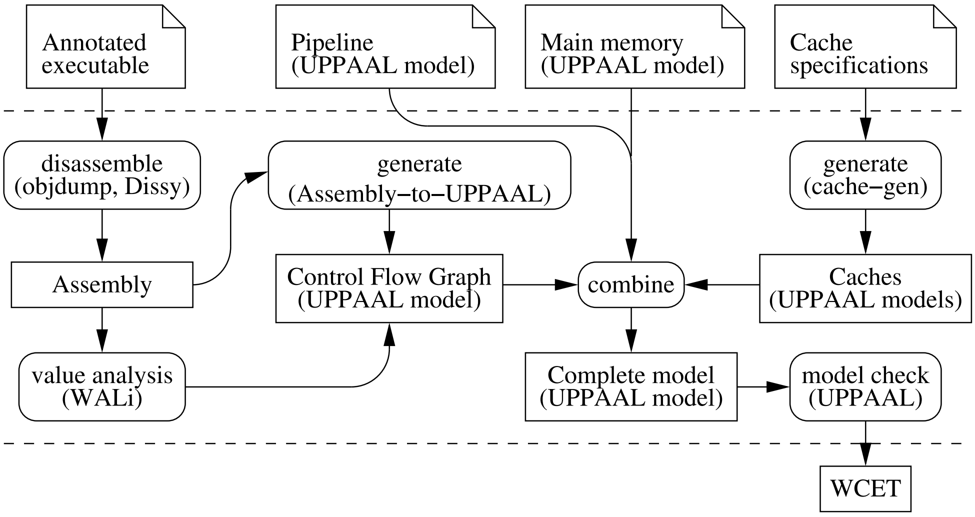


Figure 1 Overview of the prototype implementation of METAMOC. The top row shows required inputs. The executable (annotated with loop bounds) is the only user input, whereas the other inputs are platform specific models developed by specialists or hardware vendors. The output is a WCET estimate for running the executable on the hardware platform. Rounded and rectangular boxes represent actions and objects, respectively.

Modern processors utilise techniques such as caching and pipelining, which increase the average number of operations that can be executed per time unit. Since these techniques are also found in many processors intended for embedded devices, such as members of the widely deployed ARM7 and ARM9 families, a modern WCET analysis method must take them into account to be useful. The use of model checking in METAMOC provides a modular approach for dealing with these techniques: the model to be analysed comprises an abstract model of the program, and similarly for the component models for the hardware platform, which include caches, pipelines and memories. Thus, WCET analysis of a platform with, say, a new pipeline component only requires a model for the new component.

The paper is organised as follows. Section 2 provides a brief introduction to the model checker UPPAAL [2] and its extensions to timed automata (TA). Section 3 describes the models used in METAMOC for hardware components and programs, and in which ways they interact. The modularity of the method is demonstrated through support for the platforms ARM7, ARM9 and ATMEL AVR 8-bit. Section 4 details a number of experiments, which evaluate the applicability and performance of METAMOC. The experiments are conducted using a suite of WCET benchmark programs from Mälardalen Real-Time Research Centre[2]. Section 5 gives an overview of related work. Section 6 concludes the paper and presents possible directions for future work.

2 The UPPAAL Model Checker

UPPAAL [2] is a model checker for real-time systems which, besides the verification engine, features a state-of-the-art graphical user interface for modelling, simulation and verification. This section gives a brief introduction to UPPAAL models, which are used as the model

[2] `http://www.mrtc.mdh.se/projects/wcet/home.html`

formalism in METAMOC.

Systems in UPPAAL are modelled as a network of timed automata (TA), that is a set of finite automata and a set of free-running clocks that can be reset. The TA are extended with a number of features to ease modelling. *Binary synchronisation channels* enable a TA having an edge labelled `name!` to synchronise with another TA having an edge labelled `name?`, i.e. they follow the edges together in one transition. If several pairs are possible, a pair is chosen non-deterministically. *Urgent channels* dictate that synchronisations must be carried out immediately whenever possible, i.e. a time delay must not occur. Another case where a time delay must not occur is when one or more of the TA are in a location marked as *committed*. *Priorities* can be assigned to TA, such that a transition in a TA is enabled only if no transitions in any higher priority TA are enabled.

In addition to the control-flow primitives, UPPAAL models can contain a number of discrete-valued variables (and arrays of variables) and a number of C-like functions that can access and update those variables. These functions can be used as guards on transitions, or invoked when an transition is taken to update variables.

Properties to be verified for the systems are formulated in a logic inspired by timed computation tree logic (TCTL). Besides standard TCTL, UPPAAL provides a special `sup` property for finding the supremum of a clock. For example, the property "`sup: cyclecounter`" causes UPPAAL to determine an upper bound for the clock `cyclecounter`. This exact property is used by METAMOC. For an introduction to TA model checking and TCTL see [1].

3 Modelling Hardware Components and Programs

As is clear from Figure 1, METAMOC is centered around a number of models. In this section we explain the ideas behind the models and how they fit together. Starting in the upper left corner of Figure 1, the method takes as input an executable annotated with loop bounds. The executable is disassembled using the tools objdump and Dissy[3], and the resulting assembly code is given as input to a generator and a value analysis. The generator creates a control flow graph (CFG) from the assembly code, in the form of a UPPAAL model, which is annotated with results from the value analysis. Besides the executable, the method takes as input a pipeline model, a main memory model and some cache specifications. The latter are given as input to another generator, which creates cache models. Finally, the four models are combined and model checked, resulting in a WCET estimate for running the executable on the hardware platform. The CFG generator, the value analysis, the cache generator and the combine tool have been written for the prototype implementation by the authors of this paper and are released as open source.

We use a prototype implementation of METAMOC for a simplified ARM920T processor[4] as a continuing example in this section. The ARM920T processor is a member of the ARM9 family, which features an ARM9TDMI processor core[5], separate instruction and data caches, a memory management unit (MMU), and a bus interface for connecting main memory. We have modelled the core and the caches of the processor together with a simple main memory. The MMU and the bus interface are not modelled, and we have modelled least recently used (LRU) caches rather than first in first out (FIFO) caches, as FIFO caches

[3] `http://www.gnu.org/software/binutils/` and `http://code.google.com/p/dissy/`

[4] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf`

[5] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf`

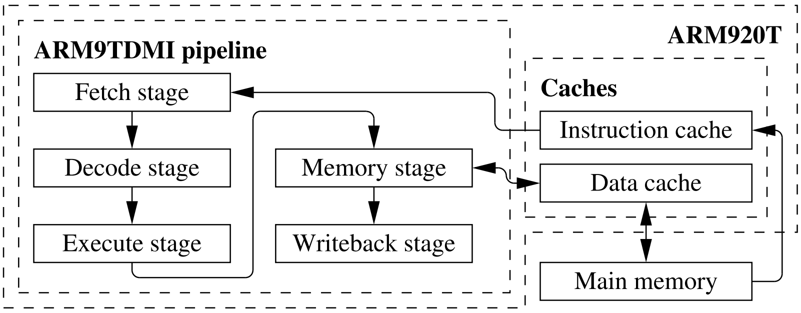


Figure 2 Communication between components in the ARM920T and the main memory.

cause timing anomalies [3]. Section 3.4 discusses the problem with FIFO caches and how they can be handled by METAMOC. The core implements the ARM instruction set v4T and contains a five stage pipeline with the stages fetch, decode, execute, memory and writeback. Communication between components in the model is illustrated in Figure 2. In order to demonstrate the modularity of the method we have utilised the ARM920T implementation to rapidly create implementations for the processors ARM7 and ATMEL AVR 8-bit. This process is detailed in Section 3.5.

In the following, a program is understood as a low-level machine executable representation, which has been disassembled to human readable assembly. The WCET of a program depends heavily on the hardware platform it is executed on, which explains why it is necessary to do the analysis at the lowest level; it is only at this level that enough information is present to determine the exact execution behaviour.

3.1 Modelling Pipelines

A pipeline is the part of a processor responsible for the execution of instructions. A pipeline divides the execution of an instruction into a number of parallel stages, in order to increase the average pace of execution. The five stages found in the ARM9TDMI core are illustrated in Figure 2. The fetch stage fetches instructions from main memory through the instruction cache. The decode stage determines the instruction type and the involved registers and prepares the needed values for the execute stage. The execute stage performs the actual arithmetic or logical computation. The memory stage accesses main memory through the data cache. Finally, the writeback stage writes computed values back into the registers. Each instruction flows through all stages, staying at least one cycle in each stage.

The parallel nature of a pipeline matches the parallel nature of a UPPAAL model, making the modelling of a pipeline in METAMOC a straight-forward process. Figure 3 shows a sketch of the UPPAAL model for the ARM9TDMI pipeline. The model contains an automaton for each stage in the pipeline. Progress in the model is forced by declaring all synchronisation channels as urgent, and time is bounded using a committed location in the writeback automaton. The non-determinism arising from the automata combinations is limited using priorities, since all combinations will result in the same state before time is allowed to progress. The simulation ensures that a safe overapproximation of the execution time is found. For example, since branch instructions are evaluated in the execute stage in hardware and in the program model in METAMOC, special handling is required. Even though the hardware flushes the fetch and decode stages in case of a branch to a non-consecutive address, the instruction cache has been affected, and we imitate this effect in METAMOC by having the fetch automaton perform two fetches without moving the fetched

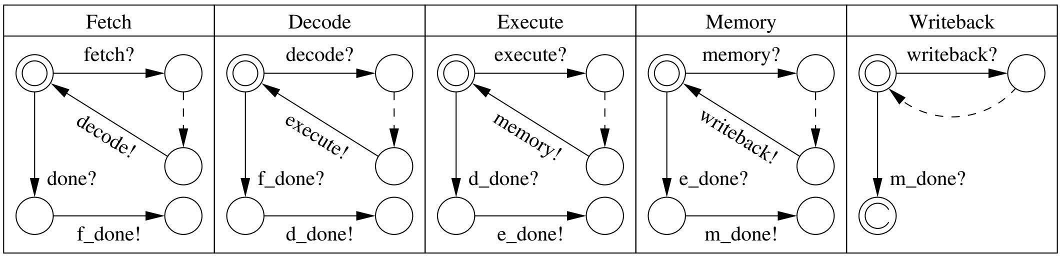


Figure 3 Sketch of the UPPAAL model for the pipeline in the ARM9TDMI processor core.

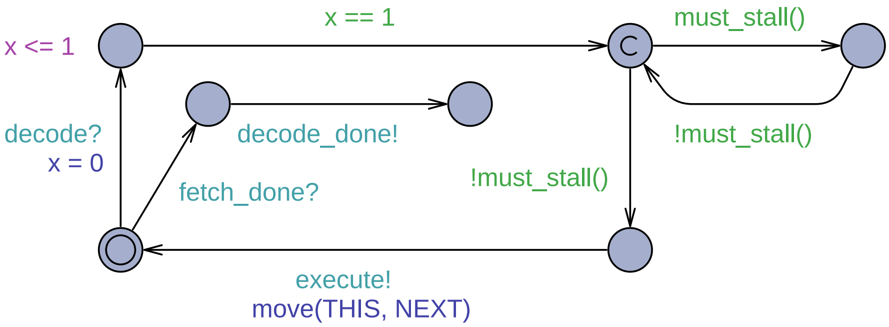


Figure 4 UPPAAL model for the decode stage in the ARM9TDMI pipeline. The model uses the local clock x, the functions move and the function must_stall. A single location is marked as committed with a C.

instructions further on in the pipeline.

Another example is pipeline stalls, which are handled in the decode automaton. The automaton initially delays for one cycle. Then, if the current instruction depends on data being loaded by the memory stage or data being shifted or sign extended by the writeback stage, it stalls until the data is ready. Consider the instructions:

```
LDR R0, [R1]     # Load R0 with the word pointed to by R1
ADD R2, R0, R1   # Store the sum of R0 and R1 in R2
```

The sum cannot be computed before the value of R0 is available, and the second instruction must therefore stay in the decode stage until the load has finished in the memory stage. The actual UPPAAL model for the decode stage is shown in Figure 4. Pipeline stalls are documented by four examples in the reference manual for the core, however, the manual does not guarantee that the examples are exhaustive. The pipeline model in METAMOC handles the four examples cycle-accurately.

To further validate our pipeline model, we have used it to calculate the number of cycles for executing some small, single-path programs from the Mälardalen WCET benchmarks and compared these cycle counts to results from the ARMulator emulator[6], assuming only cache hits. The cycle counts are comparable (with our estimates erring on the safe side), e.g. fibcall gives 407 vs. 415. It should be noted that ARMulator does not give any definite guarantees regarding cycle-accuracy[7], which means the cycle counts can only be used for approximate comparisons.

[6] http://infocenter.arm.com/help/topic/com.arm.doc.dui0058d/DUI0058.pdf
[7] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4106.html

An important property of our simplified model of the ARM920T processor is that it is free of "timing anomalies", as its pipeline is in-order [4], and it is modelled with LRU caches. If a processor has timing anomalies, it means that the local worst-case might not lead to the global worst-case. For instance, a cache hit rather than a cache miss might lead to a longer overall execution time. The absence of timing anomalies makes it convenient to find overapproximations, as the local worst-case can be used. Alternatively, if presented with a processor with timing anomalies, additional non-determinism in the model might be used to explore all local possibilities.

3.2 Modelling Caches

Another feature for improving the average execution pace is caching. The basis of caching is the principles of locality. Caches improve the pace greatly, since a main memory access might take e.g. 33 cycles while a cache access typically only requires a single cycle. A cache is divided into sets, where each block from main memory can reside in precisely one of these sets. Each set is divided into lines, also called "ways". A memory block can be stored in any of the lines in the set that it can be cached in. When a memory access occurs, eviction of a line in a cache set might be required, since all lines might be occupied. If that is the case, a replacement policy is used to determine which line to evict.

The ARM920T processor has separate instruction and data caches. Both are 16 KB, 64 way associative, have eight words (i.e. 32 bytes) per line, support the write-through and write-back write policies, and support the pseudo-random and FIFO replacement policies. As mentioned above, we consider an LRU policy in this paper. The set for a byte at address x is determined by $(x \mathbin{\&} ((ns-1) \ll log_2(ls))) \gg log_2(ls)$, where ns is the number of sets, ls is the line size in bytes, &, $\ll$ and $\gg$ are bitwise AND and SHIFT operators. This expression, slightly modified, is part of the cache models.

In order to add caching to the pipeline model, each cache is modelled as a UPPAAL model, simulating a cache hit by delaying for one cycle and a cache miss by synchronising with the UPPAAL model for main memory, which delays the appropriate number of cycles. The cache model has to keep track of which memory blocks are currently in the cache. It does so by storing an array of 512 addresses, as there are 512 lines. Cache hits are determined using this array, and the cache replacement policy is implemented as functions.

3.3 Modelling Programs

The program is modelled as a data-insensitive CFG of the program, which communicates with the fetch stage of the pipeline. Figure 5 shows a simplified example of a program with two functions: `main` and `foo`. All programs have a `main` function, which is where the execution starts. Function calls are simulated by transferring control to the function automaton and transferring control back to the call-site when the function returns. This is illustrated in Figure 5 by synchronisation over the channels `fooCall` and `fooReturn`. Bounded recursion is supported, albeit only through manual modification of the generated models. Loops are handled using loop counter variables that ensure that a back-edge of a loop can only be taken the specified number of times.

In order to reduce the amount of non-determinism in the program model, it is determinised using a simple rule: executing more code increases the execution time. Concretely, this means that loops are iterated the maximum number of times, and that forward branches are sometimes ignored. For example if a forward branch skips over a number of instructions it is ignored, as following it will only lead to less code being executed. Before ignoring such

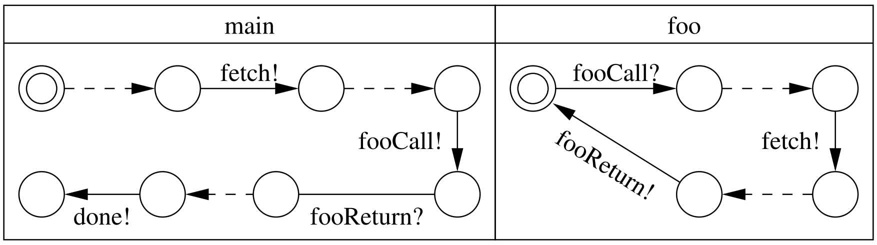


Figure 5 Sketches of the UPPAAL models for the functions `main` and `foo`.

a forward branch, the effects of the pipeline still have to be considered as if the jump is sufficiently small it might be the worst-case to take the branch, flushing the pipeline. This form of determinisation is not safe in the presence of timing anomalies.

The program CFG is annotated with the memory addresses accessed, determined statically using a value analysis. We have implemented a precise, inter-procedural constant-propagation value analysis using weighted push-down systems (WPDSs) [7] and loop unrolling. For brevity reasons we omit the details on the value analysis.

3.4 Handling Timing Anomalies such as FIFO Caches

FIFO caches cause domino effects [3], which are a type of timing anomaly. A domino effect is where the iteration of a loop body gives rise to different states of a hardware component, e.g. the pipeline or the caches, without convergence. The consequence is that it is not safe to unroll the loop any number of times and assume that state leads to the global worst case. Concretely for METAMOC this means the maximum number of loop iterations cannot be assumed, as if the bound is n, iterating the loop m times, where $m < n$, might lead to a hardware state that becomes slower overall than iterating the loop n times. By changing the guards on transitions in the program CFG that exit a loop to be more lenient, we can introduce non-determinism to explore all possible iterations of loops and thereby handle cache replacement policies that cause domino effects. Similarly, the determinisation described in Section 3.3 must be disabled for the analysis to be sound. However, the added non-determinism often results in state space explosion making the program unanalysable.

3.5 Support for ARM7 and ATMEL AVR 8-bit

Inspired by the WCET Tool Challenge 2008[8] we have implemented METAMOC for the ARM7TDMI processor core[9]. The core has three pipeline stages: fetch, decode and execute. The execute stage covers the actions performed by the execute, memory and writeback stages in the ARM9TDMI. Since the ARM9TDMI model could be reused extensively, and since both cores implement the v4T instruction set, we were able to create the ARM7TDMI model in less than a man-week.

To show that other popular embedded processors can be supported as well, we have implemented support for the ATMEL AVR 8-bit processor. It took approximately one man-week to implement the support and only required adding a new pipeline, creating support in Dissy for the AVR architecture and slightly generalising the CFG generator. Since the processor has no caches, no value analysis is performed.

[8] `http://www.mrtc.mdh.se/projects/WCC08/`

[9] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf`

4 Experiments

To evaluate the applicability and performance of our method, we evaluate it on a number of WCET benchmark programs from the Mälardalen Real-Time Research Centre. We compile the programs using a cross-compiling GNU C Compiler (GCC)[10]. The model generation is done on a 2 GHz Intel Core 2 Duo processor with 4 GB of RAM, and the model checking is done on a Dell PowerEdge 2950 with two 2.5 GHz Intel Quad Core Xeon processors and 32 GB of RAM. The UPPAAL settings for all runs are: depth-first search, aggresive state-space reduction and 64 MB hash table size.

We have manually annotated all loops in the programs with loop bounds. In addition we have promoted a few local variables to the global scope to sidestep GCC's translation of large local arrays into data segments with specialised initialiser code. We have discarded programs that either use floating point operations, do register-indirect jumps, or do not compile. GCC inserts software floating point routines, which we could analyse given an estimation of the routines' loop bounds—these are hard to estimate though, without thorough manual analysis. Out of 35 programs, this resulted in 21 programs[11] for the ARM architecture and 19 programs for the AVR architecture[12].

METAMOC has many parameters that can be adjusted for different trade-offs between precision, memory and analysis time: the compiler optimisation level, the amount of heuristic determinisation and manual annotation of the models, the level of hardware detail modelled and model checker options (specifically state space reduction techniques). To demonstrate the modularity of the method we have tested three different ARM9 configurations in order of increasing precision: with no caches (always assuming that main memory is accessed), with only an instruction cache, and with both an instruction cache and a data cache. Our value analysis is only used when the data cache is enabled. The improvements gained by using more precise models can be seen in Figure 6a, while the increase in analysis time can be seen in Figure 6b. We have omitted the benchmarks for the ARM7 architecture as the results are very similar to the ARM9 results.

Our applicability results are presented in Table 1, together with the analysis times in Figure 6b. For the ARM9 we are able to provide WCETs for all 21 benchmarks. The adpcm program results in state space explosion when enabling any caches. The ndes program is only analysable with an instruction cache with 128 cache lines. When both caches are enabled, we manually have to modify the models for three of the benchmarks: compress has a small syntactical error due to deep loop nesting; and for matmult and bsort100 the number of data cache lines modelled concretely must be reduced from 512 to 128 and 64, respectively (which amounts to editing a constant in the model editor, due to the modular design). Without this manual modification, UPPAAL runs into its 4 GB memory limit.

More AVR benchmarks suffer from state space explosion than ARM benchmarks, primarily due to the ARM architecture having support for predicated (conditional) execution of all instructions, thus reducing the number of distinct paths through the program.

The analysis times are all within 42 mins., with the average across all configurations and benchmarks being 100.47 secs. Details of the benchmarking are available at the METAMOC

[10] For ARM: GCC 4.1.2, with the options `-O2 -g -fno-builtin -fomit-frame-pointer`. For AVR: GCC 4.3.3, with the options `-O2 -g -fno-builtin -fno-inline -fomit-frame-pointer -mmcu=avr5`

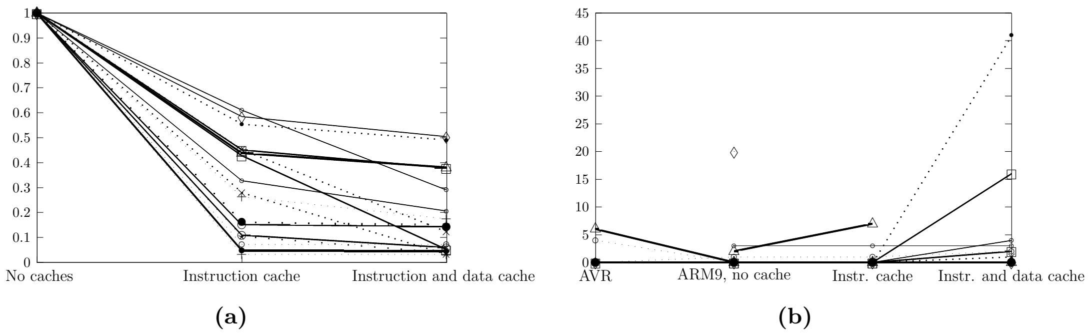
[11] adpcm, bs, bsort100, cnt, compress, crc, edn, expint, fac, fdct, fibcall, fir, insertsort, janne_complex, jfdctint, matmult, ndes, ns, nsichneu, prime, ud.

[12] The same as for the ARM, except bsort100 and nsichneu, which failed compiling due to the resulting program image being too large for the AVR.

ARM9 w. LRU caches, 21 benchmarks	
Analysable without caches	21
Analysable with instruction cache	20
Unanalysable, state space explosion	1
Manual modification of instruction cache size	1
Analysable with data and instruction cache	19
Unanalysable, state space explosion	2
Manual modification of data cache size	2
Manual syntax fix of model	1

ATMEL AVR 8-bit, 19 benchmarks	
Analysable	16
Unanalysable, state space explosion	3

Table 1 How many programs were analysable, and reasons for failure.



(a)

(b)

Figure 6 (a) Improvement in WCET estimate by precisely modelling the different caches on the ARM9. The average improvement in WCET estimate is 72.2% by modelling the instruction cache, and 82.3% by modelling both caches. (b) Analysis times for the different configurations in minutes. The average successful analysis times are, respectively: 52.46 secs., 80.76 secs., 27.84 secs. and 235.32 secs.

website, including the actual WCET estimates and UPPAAL models generated.

We have experimented with modelling an ARM9 with a FIFO instruction cache, disabling the determinisation described in Section 3.4 as this is unsound in the presence of timing anomalies, but our results are very inconclusive. Some benchmarks can be handled almost as efficiently by using different optimisations in the model checker, such as the "convex hull" overapproximation technique, while others suffer from massive state space explosion. Convex hull collapses states that only differ in their clock valuations into one overapproximated state for further exploration. The efficiency of this depends very much on the search order, but this cannot be managed at such a detailed level currently. The handling of timing anomalies is an area of future work.

5 Related Work

Using model checking for determining worst-case execution times (WCETs) is a debated approach. In [10] it is claimed that model checking is not suitable for WCET analyses, however, in [6] it is shown that model checking can actually improve WCET estimates for hardware with caching. In this paper we show that model checking can be used for WCET analysis for a simplified model of a real-world, modern processor—and with good results and performance.

Cache analyses can generally be sorted into abstract and concrete cache analyses. In the former, a model state covers a number of concrete hardware states that are similar in some way. In the latter, a model state corresponds to a particular, concrete hardware state. The common model for abstract cache analyses as in [5] has the advantage of being space efficient, with a trade-off of precision.

The pipeline analysis typically uses an abstract model of the pipeline to take its impact on the execution into account [5]. The pipeline analysis should be able to handle unknown memory values. They might lead to non-determinism, as it might be impossible to deduce a reasonable overapproximation. For this reason, abstract pipeline states are traditionally represented as a set of concrete pipeline states [8]. Recent work has looked into using binary decision diagrams (BDDs) to represent abstract pipeline states [12]. The work presented in this paper is conceptually similar but the standard reduction techniques of the model checker is used. Using real-time model checking should be more resilient to the memory delay than the method presented in [12], as delays of any length results in a single state, without intermediate states for each time unit.

For the path analysis, implicit path enumeration technique (IPET) and integer linear programming (ILP) have been combined in several tools [11, p. 42]. In [9], a path-based method is presented and has been implemented as an alternative to IPET in the SWEET tool. The method is more effective than previous path-based methods. Furthermore, path-based methods explore a path explicitly which, in contrast to IPET, could make debugging and infeasible path pruning easier. The path analysis presented in this paper is a simple exploration of the CFG of the program, with pruning of paths which cannot lead to the worst-case behaviour, but no pruning of infeasible paths.

6 Conclusion and Future Work

The optimisation features of modern processors, such as caching and pipelining, make it difficult to determine safe and tight WCETs. Our method, METAMOC, is a modular and easily retargetable approach for determining WCETs for programs running on hardware platforms featuring caching and pipelining, but no timing anomalies. In order to evaluate the method, a prototype implementation has been made for a simplified model of the ARM9 architecture, a typical processor for embedded systems. To show the modularity of the approach, the initial prototype has been extended with support for the ARM7 and ATMEL AVR architectures.

The prototype has been benchmarked to test its performance and general applicability. The experiments additionally show that much tighter WCET estimates are found when taking instruction caching into account: up to 96% tighter estimates, and 72.2% on average. Also considering the data cache increases the average to 82.3%. When taking both caches into account, the average analysis time is just under four minutes. For the ARM9 architecture, WCET estimates are given for all benchmarks, but requiring manual tweaking in four cases. For the ATMEL AVR three programs are unanalysable due to the model checker running out of memory.

Future work includes improving the model checker technology, and thereby being able to handle timing anomalies. We speculate that our models will parallelise very efficiently, as paths seem to be quite independent (especially when including caches). Distributing the model checking across more hosts will allow us to use much more memory, thereby allowing the analysis of larger programs. Exploiting the structure of our models in order to summarise the effects of long deterministic chains, e.g. basic blocks, into single steps should also help. Seeing that abstract caches seem to give a good trade-off between precision and performance, adding support for abstract caches would be interesting. Finally, rather than being data-insensitive, we would like to incorporate some form of flow facts into the program model. We already support this in some form, by allowing the user to manually annotate the program model, but it would be more beneficial if some flow facts were deduced automatically.

Acknowledgements The authors would like to thank the anonymous reviewers for their very insightful feedback.

References

1 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, first edition, 2008.

2 Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM-RT*, number 3185 in LNCS, pages 200–236. Springer, 2004.

3 Christoph Berg. PLRU Cache Domino Effects. In Frank Mueller, editor, *WCET*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

4 Jakob Engblom and Bengt Jonsson. Processor Pipelines and Their Properties for Static WCET Analysis. In *Embedded Software*, volume 2491 of *LNCS*, pages 334–348. Springer, 2002.

5 Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *LCTRTS*, ACM SIGPLAN, pages 37–46, 1997.

6 Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 334–347. Springer, 2004.

7 Thomas Reps, Akash Lal, and Nick Kidd. Program Analysis using Weighted Pushdown Systems. In *FSTTCS*, volume 4855 of *LNCS*, pages 23–51. Springer, 2007.

8 Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *LCTES*, pages 35–44, New York, NY, USA, 1999. ACM.

9 Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *CASES*, pages 132–140, New York, NY, USA, 2001. ACM.

10 Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 309–322. Springer, 2004.

11 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

12 Stephan Wilhelm and Björn Wachter. Symbolic State Traversal for WCET Analysis. In *EMSOFT*, pages 137–146, New York, NY, USA, 2009. ACM.

Precomputing Memory Locations for Parametric Allocations*

Jörg Herter[1] and Sebastian Altmeyer[1]

1 **Saarland University**
Saarbrücken, Germany
{jherter, altmeyer}@cs.uni-saarland.de

Abstract

Current worst-case execution time (WCET) analyses do not support programs using dynamic memory allocation. This is mainly due to the unpredictability of cache performance introduced by standard memory allocators. To overcome this problem, algorithms have been proposed that precompute static allocations for dynamically allocating programs with known numeric bounds on the number and sizes of allocated memory blocks. In this paper, we present a novel algorithm for computing such static allocations that can cope with parametric bounds on the number and sizes of allocated blocks. To demonstrate the usefulness of our approach, we precompute static allocations for a set of existing real-time applications and academic examples.

1998 ACM Subject Classification Performance Analysis and Design Aids

Keywords and phrases WCET analysis, cache analysis, dynamic/static memory allocation

Digital Object Identifier 10.4230/OASIcs.WCET.2010.124

1 Introduction

In modern embedded hardware, caches are used to bridge the increasing gap between processor speed and memory access times. For a timing analysis striving to derive tight bounds on a program's worst-case execution time these caches impose additional challenges. Such an analysis may not conservatively assume each memory access to be a cache miss without risking to be overly imprecise as turning off the cache completely may lead to a thirty fold increase of the execution time [5]. Also, lower bounds on the number of cache hits are not necessarily leading to tight bounds due to *timing anomalies* [7]. Timing anomalies denote situations where local worst-case behavior, i.e. a cache miss, does not always lead to global worst-case behavior. Hence, a statically more predictable cache behavior enables the derivation of tighter bounds on the programs worst-case execution times.

In the presence of dynamic memory allocation, WCET analyses fail to determine precise time bounds due to the unpredictability dynamic memory allocation inflicts on the cache behavior. General purpose dynamic memory allocators strive to cause little fragmentation and neither provide guarantees about their own worst-case execution time, nor do they provide information about the cache set mapping of the memory addresses they return. The cache set that a dynamically allocated memory block is mapped to is therefore statically not predictable. Consequently, WCET analyses can, in general, not classify accesses to dynamically allocated memory as cache hits or cache misses. Additionally, the processes of dynamic memory allocation as well as deallocation pollute the cache

*This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 124–135
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

themselves. Memory allocators manage free memory blocks in internal data structures which they maintain and traverse during allocation and deallocation operations. These unpredictable traversals of internal data structures result in unpredictable influences on the cache.
To circumvent these problems, programmers revert to static memory allocation within hard real-time programs. However, often dynamic memory allocation has advantages over static memory allocation. To enable dynamic memory allocation for hard real-time applications, Herter and Reineke proposed an algorithm to statically precompute memory addresses and replace calls to the memory allocator by sequences of fixed addresses [3]. This way, programmers can use dynamic memory allocation to alleviate the task of efficiently reusing memory, while the program itself can—after an automatic transformation—be analyzed by current WCET analyses.
However, the algorithm proposed in [3] is limited to programs that contain only loops bounded by a numerical value. Additionally, the requested block sizes—or at least tight bounds on these—need to be statically known. In this paper, we propose a novel algorithm that can cope with parametric bounds on the number of possible loop iterations and requested block sizes.

Figure 1 visualizes the application area for the proposed algorithm. We start with a program using dynamic memory allocation that is intended to be deployed in a hard real-time setting. However, due to the unpredictability that dynamic memory allocation inflicts on the cache behavior of the program, no tight bounds for the program's worst-case execution can be determined using current WCET analyses.

In a first phase, we apply a static program analysis to compute liveness information for the dynamically allocated objects. This information together with user supplied loop and recursion bounds is used as input for the proposed algorithm for precomputing static memory addresses. In the second and final phase, we replace calls for memory allocation by functions that return a sequence of precomputed addresses. Using fixed memory addresses, calls to `free` become obsolete and can simply be removed. This phase yields a modified program in which the memory addresses of all objects are statically known. For such a program, current WCET analyses can compute tight bounds on its worst-case execution time.

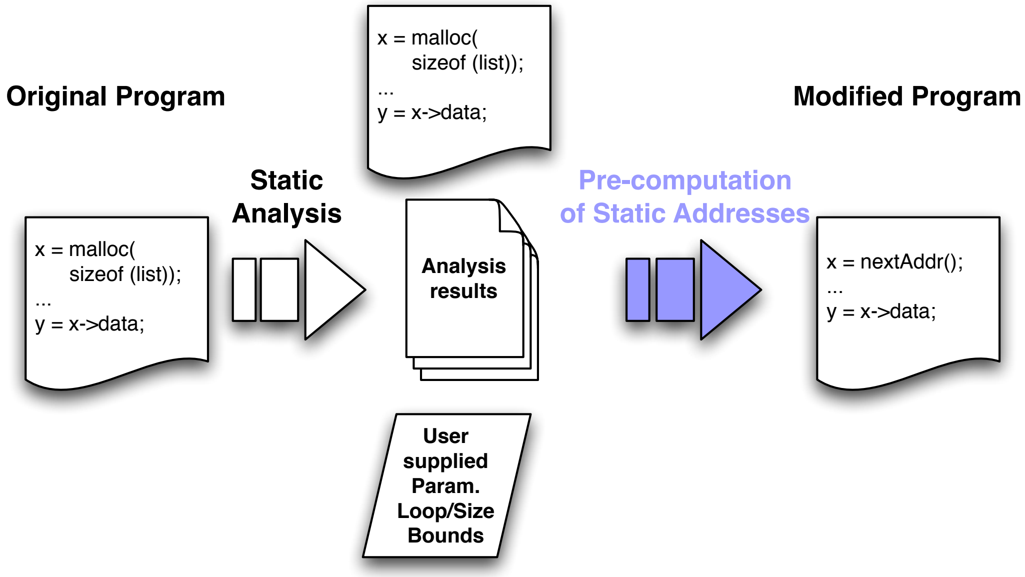


Figure 1 Field of application for the proposed algorithm

The remainder of this paper is organized as follows. Section 2 briefly reviews related work. In Section 3, we formally describe a program's memory allocation behavior. The algorithm presented in Section 4 uses such a formal description to statically precompute memory addresses for the allocated memory blocks. In Section 5, we present results obtained from precomputing memory allocations for a set of academic and real-life programs.

2 Related Work

There are two other approaches to make programs that dynamically allocate memory more analyzable with respect to their WCET. In [4], Herter et al. propose to utilize a predictable memory allocator to overcome the problems introduced by standard memory allocators. Schoeberl proposes different (hardware) caches for different data areas [10]. Hence, accesses to heap-allocated objects would not influence cached stack or constant data. The cache designated for heap-allocated data would be implemented as a fully-associative cache with an LRU replacement policy. For such an architecture it would be possible to perform a cache analysis without knowledge of the memory addresses of the heap-allocated data. However, a fully-associative cache, in particular with LRU replacement, is limited in size due to technological constraints. Those two approaches have the advantage not to rely on tight bounds on the number of allocation requests and the sizes of requested memory blocks.

An algorithm to statically precompute memory addresses and replace calls to the memory allocator by fixed addresses for programs in which all occurring loops and requested sizes can be (tightly) bounded by a numerical value was proposed in [3]. This approach, although applicable to a smaller class of applications, has the advantage to yield entirely static allocations, removing memory allocation procedures completely.

3 A Formal Model for Memory Allocations

To describe a program's memory allocation behavior, we start by collecting all allocation sites, i.e. occurrences of `malloc` within the program, in a set M. Per assumption, we know how often each allocation site can be reached during program allocation as loop and recursion bounds are known, at least parametrically. Hence, we introduce a further set

$$U = \bigcup_{m \in M} \{u_m\}$$

where $u_m \in \mathbb{N} \cup P$ is an upper bound on how often allocation site m may be reached, i.e. how often this function call may be invoked. P denotes the set of parametric loop and recursion bounds. For each allocation site m, we construct a function f_m such that $f_m(i)$ evaluates to the size of the memory block requested the i-th time allocation site m is reached. These sizes may be over-approximated by intervals. And consequently,

$$A = \bigcup_{m \in M} \left\{f_m : \mathbb{N}^{\leq u_m \in U} \mapsto \mathcal{I}_m\right\}$$

where $\mathcal{I}_m$ is a set of intervals, is the set containing functions describing the allocation requests for all allocation sites. Now, the set

$$R = \left\{(m, i) \mid m \in M \land i \in \mathbb{N}^{\leq u_m \in U}\right\}$$

contains all allocation requests that may occur during program execution.

For precomputing feasible memory allocation we also need to know which allocated blocks have overlapping lifetimes. Liveness information for dynamically allocated memory blocks can safely be over-approximated using shape analysis [9]. We assume this information to be available and encode it in a conflict function

$$\mathcal{C} : 2^R \mapsto \{0, 1\}$$

that evaluates to 1 iff its argument requests at least two memory blocks with overlapping lifetimes.

When precomputing static memory addresses for originally dynamically allocated memory blocks, we may not want to ignore cache set mappings completely. Hence, we are presented two options. We may strive for cache set mappings leading to further improved predictability of the program. This, however, requires us to be aware of many details of the cache analysis applied to the transformed program. Basically, to be able to decide which cache set mapping may enable a subsequent cache analysis to classify the largest number of memory accesses as hits or misses calls for knowing the exact analysis algorithm. The second option would be to strive for good cache performance such that the risk is reduced that the statically precomputed addresses decrease program performance. In order not to rely on assumptions about subsequently applied analysis techniques or to be restricted to specific analyses, we favor the second option.

Unfortunately, under the assumption that $P \neq NP$, one cannot efficiently approximate an optimal placement of objects in memory that reduces the number of cache misses [8]. However, Chilimbi et al. showed that simply trying to place objects that are likely to be accessed contemporaneously next to each other in memory achieves significant increases in performance [1]. To exploit this heuristics, we construct a bias function

$$\mathcal{B} : (R \times R) \mapsto \{0, 1\}$$

such that $\mathcal{B}(r_1, r_2)$ evaluates to 1 iff the block requested in r_1 is likely to be accessed prior to the one requested in r_2. How can we statically obtain information about what objects will be accessed contemporaneously during program execution? Chilimbi's work relied on the user to provide this information. While this yields the most precise information in most cases, we can also approximate object access behavior using shape analysis [9]. We say that two objects o_1 and o_2 are likely to be accessed contemporaneously if there exist field pointers between o_1 and o_2. A third, more efficient but potentially less precise way to gather this information is to apply a data structure analysis [6] together with the heuristics that objects organized in the same data structure are likely to be accessed contemporaneously.

An allocation problem is then a six-tuple

$$(M, U, L, A, \mathcal{C}, \mathcal{B})$$

where L is a set of constraints on the parameters in P.

An allocation is a feasible solution to an allocation problem of the form

$$\bigcup_{r \in R} \{(r, addr)\}$$

where $addr$ denotes the precomputed starting address of the memory block requested by r.

An optimal allocation is a feasible solution to an allocation problem such that (1) there is no other feasible solution with smaller memory consumption. And (2), considering the set of all feasible solutions with minimum memory consumption, no solution exists that places more blocks, for which the bias function $\mathcal{B}$ evaluates to 1, in memory next to each other.

Finding such optimal solutions is still at least NP-hard. Let (V, E, k) be a given instance of the k-colorability problem for a graph $G = (V, E)$. Generate the allocation problem

$$K = (V, \{1\}, \{\}, \{f : \{1\} \mapsto [1, 1]\}, C : R \times R \mapsto \{0, 1\}, B : R \times R \mapsto \{0\})$$

where B maps all arguments to 0 and C is defined s.t.

$$C(S) = 1 \Leftrightarrow \exists v_1, v_2.(v_1, v_2) \in E \wedge \{v_1, v_2\} \subseteq S$$

This transformation can be done in polynomial time and can be used to solve the k-colorability problem for G as follows. Find an optimal allocation for K and check whether less than or equal to k memory addresses are used, in which case G is k-colorable by associating each memory location with a (different) color.

We choose a heuristic approach to finding *good* solutions as striving for optimal solutions would render our technique only applicable to very small applications.

4 Algorithm

Before starting to precompute suitable memory addresses for allocated memory blocks, the algorithm transforms its input $I = (M, U, L, A, \mathcal{C}, \mathcal{B})$ as follows. Using the bias function $\mathcal{B}$, maximal ranges of allocated memory blocks that should be placed adjacent in memory are identified. These ranges are then split again into blocks, such that the sizes of the resulting *normalized blocks* are multiples of the size of a cache line. The last normalized block may be of smaller size and splitting must not occur within the bounds of an allocated memory block. This transformation yields a new allocation problem $I' = (M', U', L', A', \mathcal{C}')$ for a parametric set of normalized blocks. M' denotes now the set of maximal ranges of allocated memory blocks, or *abstract allocation sites*, while the functions collected in A' map to the sizes of the single normalized blocks of these ranges. $U' \subset \mathbb{N} \cup P'$ and L' are the accordingly updated constraints on loop/recursion bounds and parameters. We gain two advantages from this transformation step. The number of blocks considered by the algorithm is in general reduced, leading to a smaller problem instance. Also, the bias function was consumed and the algorithm does not need to respect further constraints introduced by this function.

Given the input I', the values of the parameters P' are unknown at design time. Hence, the resulting allocation scheme is parameterized in P'. To enable the algorithm to cope with this, we introduce *memory block chunks*. A chunk is a relative placement of memory blocks from different abstract allocation sites in M', such that there is no conflict within a chunk itself. Chunks are then placed sequentially in memory such that memory blocks of each abstract allocation site $m \in M'$ occur exactly u_m times. Formally, a chunk is a set of triples (m, i, o), with the intended meaning that the i-th request from abstract allocation site m is located within the chunk at relative position o. An allocation, i.e. solution to an allocation problem, contains several types of chunks and the number of occurrences of a specific chunk is parametric. By this, we reduce the computation of a solution to an allocation problem to finding an appropriate selection of chunks.

Let us consider a standard example for advantageous use of dynamic memory allocation, namely in-situ list copy. Assume, we want to copy a singly-linked list M_s to a doubly-linked list M_d with minimal memory consumption. The size of a list is determined by a parameter p. Assume, the singly-linked list is traversed once and on this traversal the visited elements are copied to newly allocated elements of the doubly-linked list. Then, the i-th element of the singly-linked list has a conflict with the j-th element of the doubly-linked list iff $j < i$. An allocation scheme with minimal memory consumption is given in Figure 2a. Note that a memory optimal allocation is not necessarily unique, neither are optimal solutions in general. Figure 2b gives a second possible mapping for our list example with minimum memory consumption.

What set of chunks would we like our algorithm to compute? With the additional constraint that the size of a chunk is determined by the size of the largest normalized memory block it contains, we would anticipate a solution set

$$S = \{\{(M_d, 1, 0)\}, \{(M_s, p, 0)\}\} \cup \bigcup_{i \in [2,p]} \{(M_d, i, 0), (M_s, i-1, 0)\}$$

Figure 3 shows these chunks and their corresponding number of occurrences. Putting these chunks consecutively in memory yields the optimal allocation scheme shown in Figure 2b.

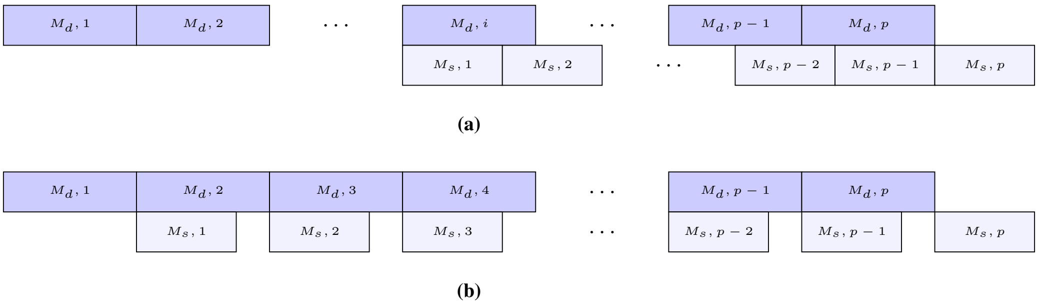


Figure 2 Allocation schemes for list-copy with minimal memory consumption

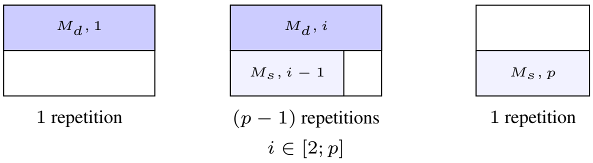


Figure 3 Allocation chunks for the list-copy example

We distinguish 2 kinds of chunks, singleton and repetitive chunks. A singleton chunk is a chunk that is generated exactly once, while multiple instances of repetitive chunks are generated. In our list example, the algorithm generates 2 singleton and 1 repetitive chunks.

Our algorithm to compute such sets of chunks for a given allocation problem works as follows. The algorithm maintains a workset of unprocessed, i.e. not yet located within a chunk, requests for normalized blocks. This set is initialized with the set

$$R' = \left\{(m, i) \mid m \in M' \land i \in \mathbb{N}^{\leq u_m \in U'}\right\}$$

While the workset is not empty, the algorithm creates singleton chunks followed by sequences of repetitive chunks. Algorithm 1 gives the pseudo code for this main routine.

Algorithm 1: Algorithm to compute a suitable set of chunks

```
Data: Problem specification $I' = (M', U', L', A', \mathcal{C}')$
Result: Allocation scheme as a set of chunks
workset = Set of requests $R$ obtained from $M'$, $U'$ and $A'$;
while $workset \neq \emptyset$ do
    createChunk(workset, true); // first chunk – unrolling
    removeProcessedRequests();
    if $workset = \emptyset$ then break;
    createChunk(workset, false); // create repetitive chunk
    computeRepetitions(); // repeat last chunk
    removeProcessedRequests();
end
```

The function createChunk creates new chunks and adds normalized blocks until no further blocks are requested for a given abstract allocation site or no further blocks can be added without either causing a conflict or exceeding the size of the chunk. The order in which blocks are added is either given by the problem specification (in the case of singleton chunks) or in decreasing order of block sizes (in the case of repetitive chunks). This order also determines the size of a chunk. Algorithm 2 gives the pseudo code for this function.

Algorithm 2: createChunk

```
Data: Set of requests $R$, boolean isSingleton
Result: Chunk
if $\neg$ IsSingleton then sortByRequestSize($R$)
for $(m, i) \in R$ do
    boolean added = true;
    while $i < u_m \wedge added$ do
        added = addRequestToChunk();
        if added then i++;
    end
end
```

Requests or normalized blocks are added to a given chunk in the following way. If the chunk is empty, the request is always added at the first position of the chunk and the size of the chunk is set to the size of the first request. Subsequent blocks are temporarily placed at position $p = 0$. In case this does not cause conflicts, the request is added and the algorithm returns true. While conflicts do occur, the subsequent request is shifted to the next position $p + 1$ until either all conflicts are solved and the requested block is added or no space in the chunk is left and the block is not added. Algorithm 3 gives the pseudo code for this operation.

```
Algorithm 3: addMallocToChunk
  Data: Request $r$
  Result: boolean added
  if chunk.isEmpty() then
      addAtZero($r$);
      return true;
  end
  int pos = 0;
  while $pos \leq ChunkSize - sizeOf(r)$ do
      If $\neg$ conflictInChunk() return true;
      pos++;
  end
  return false;
```

The problem specification may contain several parameters and also the sizes of requested blocks can be parametric. Hence, not all conditionals within the above functions may be computed directly. Only the set of parameter constraints L given as part of problem specification may be used to decide these conditions. If the set L is not sufficient to allow for deciding conditionals, we split the specification depending on the various outcomes.

For instance, a conditional

$$\text{if } p < q$$

leads to the following two problem specifications:

$$S = (M, U, L \cup \{(p < q\}, A, \mathcal{C}, \mathcal{B})$$

and

$$S = (M, U, L \cup \{(p \geq q\}, A, \mathcal{C}, \mathcal{B})$$

Hence, each time the set of restrictions on parameters, L_S, does not contain enough information to decide whether a conditional c is satisfied, we replace the current allocation problem S by two new allocation problems, S_t and S_f. In S_t, we set L_{S_t} to $L_S \cup \{c\}$, and accordingly in S_f to $L_S \cup \{\neg c\}$.

5 Experiments

We did a preliminary evaluation of our algorithm using two programs that perform an in-situ copy from one data structure to another as well as a small set of existing (hard) real-time programs taken from the MiBench benchmark suite [2].

In-Situ Copy

The memory allocation behavior of a program copying a singly-linked list to a doubly-linked list as used as an example in Section 4 can be formalized as

$$(\{m_s, m_d\}, \{p_s, p_d\}, \{p_s = p_d\}, \{f_{m_s} : [p_s] \mapsto [8, 8], f_{m_d} : [p_d] \mapsto [12, 12]\}, \mathcal{C}_{lc}, \mathcal{B}_{lc})$$

With the intended meaning that there are two allocation sites, one for the elements of the singly-linked list, one for those of the doubly-linked one. Each site can be reached at most p_s and p_d times, respectively. We know, that $p_s = p_d =: p$ as all elements are copied. $\mathcal{C}_{lc}$ is constructed, such that there are conflicts between list elements that are in-use at the same time. $\mathcal{B}_{lc}$ evaluates to 0 for all inputs, i.e. no bias is given, to prevent bias disabling the algorithm to compute memory optimal solutions. Our algorithm was able to compute for such a program description a memory optimal set of chunks as depicted in Figure 2b. With $\mathcal{B}_{lc}$ constructed such that elements adjacent within a list are to be put adjacent in memory, our algorithm computes a similar set of chunks as depicted in Figure 4. This normalization yields blocks of size 96 KB and 32 KB for the doubly- and singly-linked list, composed of 8 and 4 objects of the original lists, respectively. However, the repetitive chunk contains now two blocks from the singly-linked list together with one block from the doubly-linked one.

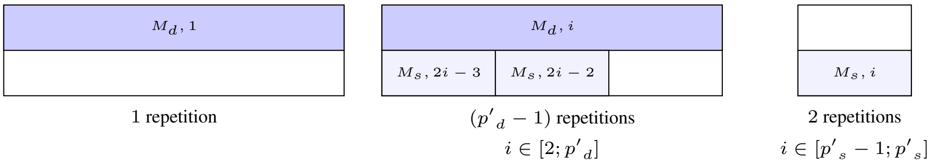


Figure 4 Allocation chunks for normalized in-situ list copy

Consider next the reverse list-copy from doubly-linked to singly-linked elements

$$\big(\{m_s, m_d\}, \{p\}, \{\}, \{f_{m_s} : [p_s] \mapsto [q, q], f_{m_d} : [p_d] \mapsto [12, 12]\}, \mathcal{C}_{lc}, 2^R \mapsto \{0\}\big)$$

with parametric sizes. At allocation site m_s blocks of size q KB are requested, at site m_d 12 KB blocks, with $4 < q \leq 12$. On this example, our algorithm computes the following solutions.

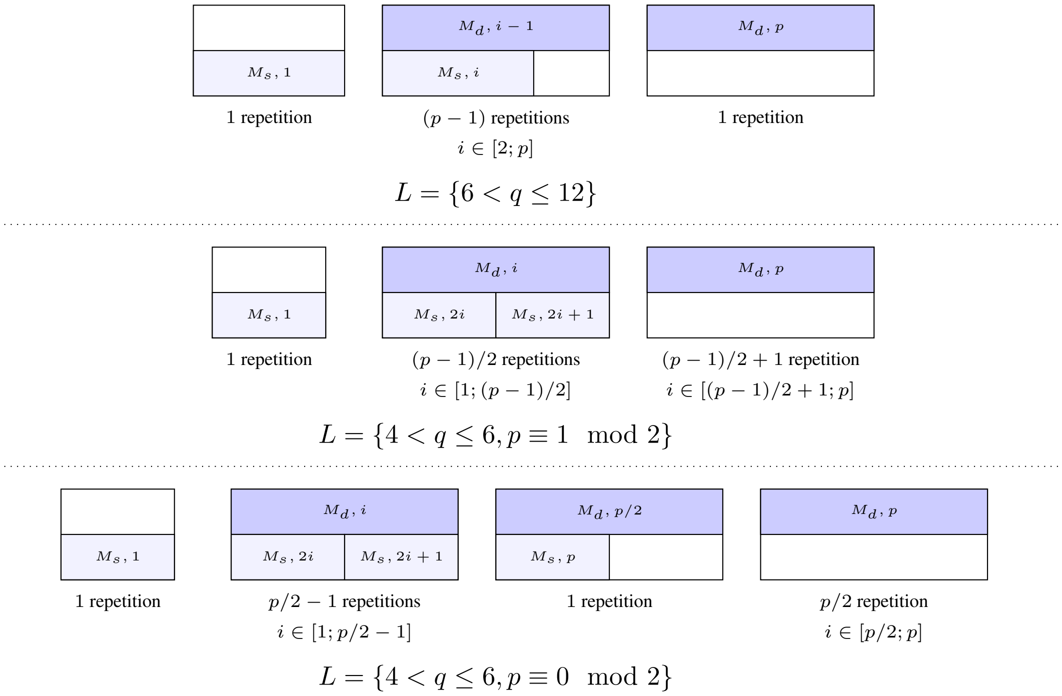


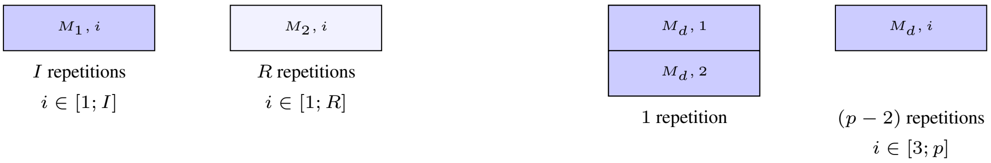
Figure 5 Allocation chunks for the reversed in-situ list-copy example

Patricia (MiBench)

A patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Patricia tries are often used to represent routing tables in network applications. This application uses patricia tries to construct a routing table. The benchmark program is formalized to

$$(M_p, U_p, \{\}, A_p, \mathcal{C}_p, \mathcal{B}_p, (R \times R) \mapsto \{0\})$$

where $M_p = \{1, 2\}$, $U_p = \{I, R\}$, $A_p = \{f_1 : \mathbb{N}^{\leq I} \mapsto [8 \cdot ml, 8 \cdot ml], f_2 : \mathbb{N}^{\leq I} \mapsto [8 \cdot ml, 8 \cdot ml]\}$, and $\mathcal{C}_p(C) = 1$. The parameter ml denotes a variable value, possibly determinable by a value analysis. As we cannot safely determine that two allocated blocks are not contemporaneously in-use, our algorithm is not able to compute a better memory allocation than the one given in Figure 6a. This application shows one limitation of precomputing memory addresses, namely that a previous analysis must be able to gather precise liveness information regarding which allocated blocks are alive at the same time.



(a) Patricia **(b) Dijkstra**

Figure 6 Allocation chunks for the Patricia and Dijkstra test cases

Dijkstra (MiBench)

Dijkstra constructs a large graph (as an adjacency matrix) and then computes the shortest paths between pairs of nodes using repeated applications of Dijkstra's algorithm. The program can be described by

$$(M_d, U_d, \{\}, A_d, \mathcal{C}_d, \mathcal{B}_d)$$

where $M_d = \{1\}$, $\{u_1\} = \{n^2\}$, $A_d = \left\{\left\{x \mapsto [16, 16] \mid x \in \mathbb{N}^{\leq n^2}\right\}\right\}$, and

$$\mathcal{C}_d(C) = \begin{cases} 1 & \text{if } \exists i, j \in C . j \in (i, (i-1) \cdot n] \\ 0 & \text{otherwise} \end{cases}$$

Here, n is the number of nodes of the constructed graph. For this application, our algorithm computed a chunk set as depicted in Figure 6b.

Susan (MiBench)

Susan is an image processing application used to determine the position of edges and/or corners within the input image for guidance of unmanned vehicles. Its allocation behavior can be formalized to

$$(M_s, U_s, \{\}, A_s, \mathcal{C}_s, (R \times R) \mapsto \{0\})$$

where $M_s = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $\bigcup_{m \in M_s} u_s = \{1\}$, $A_s = \{f_1 : \{1 \mapsto x \cdot y\}, f_2 : \{1 \mapsto 516\}, f_3 : \{1 \mapsto (14 + x)(14 + y)\}, f_4 : \{1 \mapsto 16\}, f_5 : \{1 \mapsto 4 \cdot x \cdot y\}, f_6 : \{1 \mapsto 4 \cdot x \cdot y\}, f_7 : \{1 \mapsto 4 \cdot x \cdot y\}, f_8 : \{1 \mapsto x \cdot y\}, f_9 : \{1 \mapsto 4 \cdot x \cdot y\}\}$, and

$$\mathcal{C}_s(C) = \begin{cases} 1 & \text{if } \{(1,1)\} \subsetneq C \vee \{(7,1), (8,1)\} \subseteq C \vee \{(3,1), (4,1)\} \subseteq C \\ 0 & \text{otherwise} \end{cases}$$

Here, x and y are fixed parameters determining the size of the processed images. Again, our algorithm computed a memory optimal chunk set as depicted in Figure 7.

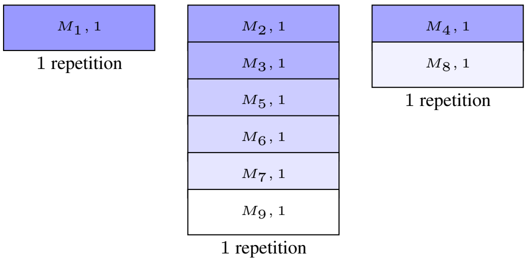


Figure 7 Allocation chunks for the Susan test case

6 Conclusions

Statically precomputing memory addresses for otherwise dynamically allocated blocks yields significant advantages. Compared to striving for predictable memory allocation, predictability of the program is increased as addresses of (heap) objects become statically known. Furthermore, with allocation and deallocation removed completely from the program, so are unpredictability resulting from cache pollution caused by allocators and the uncertain response times of (de)allocation routines removed. However, static precomputation is not applicable to all hard real-time applications. While the algorithm presented in this paper increases the class of programs that such an approach can cope with, there are still limitations. The Patricia benchmark showed that if such an approach is to preserve the main advantage of dynamic allocation, efficient memory reuse, precise information about which blocks may be allocated at the same time must be available to the algorithm. Hence, such an approach relies on precise static preanalyses to yield results that enable similar program performance compared to a program using dynamic memory allocation. Without such analyses, predictability comes at the price of overly high memory consumption. Furthermore, too little information about the relations between parameters can lead to an overly large solution set, as each time decisions cannot be made due to incomplete information, the algorithm considers two cases, splitting to two solution paths. However, given decent information about the program to transform, our algorithm showed very promising results. An automatic analysis to gather these informations as well as an exhaustive evaluation of the presented algorithm are to be tackled next.

References

1 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

2 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

3 Jörg Herter and Jan Reineke. Making dynamic memory allocation static to support WCET analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

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

5 M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. *Proceedings of Static Analysis Symposium (SAS)*, 2477, 2002.

6 Chris Lattner and Vikram Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Technical report, 2003.

7 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

8 Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *Nordic J. of Computing*, 12(3):275–307, 2005.

9 Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

10 Martin Schoeberl. Time-predictable cache organization. In *STFSSD '09: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, pages 11–16, Washington, DC, USA, 2009. IEEE Computer Society.

The Mälardalen WCET Benchmarks: Past, Present And Future

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper

School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden.
{jan.gustafsson,adam.betts,andreas.ermedahl,bjorn.lisper}@mdh.se

Abstract

Modelling of real-time systems requires accurate and tight estimates of the Worst-Case Execution Time (WCET) of each task scheduled to run. In the past two decades, two main paradigms have emerged within the field of WCET analysis: static analysis and hybrid measurement-based analysis. These techniques have been succesfully implemented in prototype and commercial toolsets. Yet, comparison among the WCET estimates derived by such tools remains somewhat elusive as it requires a common set of benchmarks which serve a multitude of needs.

The Mälardalen WCET research group maintains a large number of WCET benchmark programs for this purpose. This paper describes properties of the existing benchmarks, including their relative strengths and weaknesses. We propose extensions to the benchmarks which will allow any type of WCET tool evaluate its results against other state-of-the-art tools, thus setting a high standard for future research and development.

We also propose an organization supporting the future work with the benchmarks. We suggest to form a committee with a responsibility for the benchmarks, and that the benchmark web site is transformed to an open wiki, with possibility for the WCET community to easily update the benchmarks.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.136

1 Introduction

Bounding the Worst-Case Execution Time (WCET) of real-time software is crucial when developing and verifying real-time systems. These bounds must be safe and tight (i.e., as close to the actual WCET as possible).

WCET analysis attempts to deliver such a bound. Its techniques can broadly be categorised as follows:

- *End-to-end measurements* is the traditional approach and is used widely in industry. Test-vector generation algorithms attempt to stress the longest execution time of the program under analysis. To try and bypass any optimism, some additional margin is added to the longest recorded time and this is considered as the WCET estimate.
- *Static analysis* relies on mathematical models of the software and hardware involved. The hardware model allows the execution time of individual instructions to be gleaned. The software model represents possible execution flows. Combining these models with information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc., results in a WCET estimate. Provided that the models are correct, the WCET estimate is always safe, i.e., greater than or equal to the actual WCET.
- *Hybrid measurement-based analysis* operates similarly to static analysis, except it does not create a hardware model. Rather, it uses measurements to derive execution times of small program parts, before combining them using flow information in the WCET calculation. Although the WCET estimate is generally more accurate than that computed

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 136–146
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

by static analysis, there is a possibility of underestimation if testing has not sufficiently stressed the execution times of the small program parts.

A number of WCET analysis tools have emerged in recent years. Academic toolsets of note include: OTAWA [14], Chronos [11], SWEET [12], and Heptane [8]. Some of the developed techniques have also migrated into fully-fledged commercial tools, including: RapiTime [18], aiT [1], and Bound-T [21]. However, a comparison between these tools, and the associated methods and algorithms, requires a common set of benchmarks. The typical evaluation metric is the accuracy of the WCET estimate, but of equal importance are other properties such as performance (i.e., scalability of the approach) and general applicability (i.e., ability to handle all code constructs found in real-time systems). In summary, it is very useful to have an easily available, thoroughly tested, and well documented common set of benchmarks in order to enable comparative evaluations of different algorithms, methods, and tools.

The Mälardalen WCET benchmarks have been assembled with the above goals in mind. This paper describes properties of the existing benchmarks, including their relative strengths and weaknesses. In particular, we propose to extend the benchmarks with new types of codes, which will raise the standard for future research and development of WCET algorithms, methods, and tools. We also propose an organization supporting the future work with the benchmarks.

The rest of the paper is organized as follows: Section 2 places the Mälardalen WCET benchmarks into context by reviewing other benchmark suites on offer. Following that, Section 3 describes the WCET benchmarks and Section 4 evaluates them, presenting ideas for development of an extended version. Section 5 concludes the paper and presents future work.

2 Related Benchmarks

Benchmarking is a problem not only in the WCET community but across various computing disciplines. For this reason, the number of available benchmark suites for computer science is large. A typical goal of these benchmarks is to evaluate performance for various computing areas, for example for integer and floating-point calculations, e.g., the Drystone benchmark [3], and for stressing a system's processor, memory subsystem and compiler, e.g., the SPEC CPU2006 benchmark [2].

As an exhaustive examination of benchmarking suites for computer science is beyond the scope of this paper, this section instead relates to those which have most relevance to WCET analyses.

The goal of the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [4] is to specify benchmarks for both the hardware and software utilised in embedded systems. At the time of this writing, eight suites are available, each of which is designed to stress a particular type of workload in the embedded domain, including: automotive, digital imaging, digital entertainment, energy consumption, mobile Java applications, networking, office automation (e.g. printers), and telecommunications. All of the benchmarks are written in C or Java. A benefit of the EEMBC benchmarks is that they are continually maintained and updated, as the consortium is run as a non-profit organisation. However, the downside is that gaining access to the benchmarks requires a licence, even for academics.

Drawing motivation from this deficiency, the MiBench benchmarks [13] were proposed, which are open source in comparison and are written in C. Similarly to the EEMBC suites, MiBench splits its programs into six distinct groups: automotive, consumer, networking, office automation, security, and telecommunications. Given their strong correlation to the embedded domain, many of these benchmarks appear to be suitable candidates for WCET analysis, although they have been sparsely used in the WCET community.

In the WCET tool challenge of 2008 [22], several other benchmarks were introduced. The DEBIE-1 benchmark is a satellite application, written in C, consisting of six tasks. The main appeal of this benchmark is that it is a realistic application, having been initially supplied by Space Systems Finland Ltd, and it is shipped with a test harness (developed by Tidorum Ltd [21]) thereby easing measurement-based analyses. The other four benchmarks used, *rathijit_1* through *rathijit_4*, were provided by Saarland University, and aim to have large instruction cache and data cache footprints. The DEBIE-1 benchmark is not open source, but can be requested from Tidorum Ltd, whereas the *rathijit* applications are freely available.

PapaBench [15] is another recently proposed benchmark in the WCET community, which is based on an actual real-time application from within the avionic industry. PapaBench is a real-time embedded benchmark derivated from the software of a GNU-license UAV, called Paparazzi. Formerly driving a bi-processor AVR architecture, the application C sources have been adapted to compile under several other platforms. Similarly to the DEBIE-1 benchmark, PapaBench consists of a number of tasks and interrupts.

3 The Mälardalen WCET Benchmarks

The Mälardalen WCET benchmarks were collected in 2005 from several researchers within the WCET field. Properties of each benchmark program (which are all written in C) are listed in Table 1 and 2.

The purpose of the Mälardalen WCET benchmarks is to have a common, easily available, set of test programs for WCET methods and tools. The benchmarks includes a broad set of program constructs to support testing and evaluation of WCET tools.

The Mälardalen WCET benchmarks are available on a web page [23]. The benchmark programs are marked with the following properties: I = uses include files (i.e., uses more than one file), E = calls external library routines, S = is a single path program (no flow dependency on external variables), L = contains loops, N = contains nested loops, A = uses arrays and/or matrices, B = uses bit operations, R = contains recursion, U = contains unstructured code, and F = uses floating point calculation. The size of source code file (bytes), as well as LOC = number of lines of source code, is also provided.

There are some main categories of benchmark programs:

- Well-structured code (all benchmark programs except `duff`)
- Unstructured code (`duff`)
- Array and matrix calculations (`bs, bsort100, edn, fdct, fft1, insertsort, ludcmp, matmult, minver, ndes, ns, qsort-exam, qurt, select, st`)
- Nested loops (`adpcm, bsort100, cnt, compress, crc, edn, expint, fft1, fibcall, fir, insertsort, janne_complex, ludcmp, matmult, minver, ns, qsort-exam, select`)
- Input dependent loops (`bsort100, janne_complex, insertsort`)
- Inner loops depending on outer loops (`crc, fir, janne_complex, insertsort`)
- Switch cases (`cover`)
- Nested if-statements (`nsichneu`)
- Floating point calculations (`fft1, lms, ludcmp, minver, qsort-exam, qurt, select, sqrt, st`)
- Bit manipulation (`crc, edn, fdct, lcdnum, ndes`)
- Recursive code (`recursion`)

Program	Description	Comments
`adpcm`	Adaptive pulse code modulation algorithm.	Completely well-structured code.
`bs`	Binary search for the array of 15 integer elements.	Completely structured.
`bsort100`	Bubblesort program.	Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.
`cnt`	Counts non-negative numbers in a matrix.	Nested loops, well-structured code.
`compress`	Compression using lzw.	Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data.
`cover`	Program for testing many paths.	A loop containing many switch cases.
`crc`	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.
`duff`	Using “Duff’s device” to copy 43 byte array.	Unstructured loop with known bound, switch statement
`edn`	Finite Impulse Response (FIR) filter calculations.	A lot of vector multiplications and array handling.
`expint`	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.
`fdct`	Fast Discrete Cosine Transform.	A lot of calculations based on integer array elements.
`fft1`	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.	A lot of calculations based on floating point array elements.
`fibcall`	Iterative Fibonacci, used to calculate fib(30).	Parameter-dependent function, single-nested loop
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $(n^2)/2$ iterations (triangular loop).

Table 1 Benchmark programs (part 1)

- Automatically generated code (`nsichneu, statemate`)

3.1 Additional information provided

The web page also includes meta-data for the benchmarks: inputs for some of the benchmarks, number of loop iterations, and some types graphs. This is described in more detail in the following.

Single-path/multi-path benchmarks and inputs to the benchmarks.

The programs in the benchmark can all be run "as is", i.e., the programs contain their own inputs. This means that they execute a single path. However, most realistic programs are run with different inputs at different invocations. If the inputs can affect the control flow, the program’s WCET is usually highly dependent on inputs.

For WCET analysis, it is important to know the possible values of the input variables since these, in general, must be constrained as much as possible in order to obtain tight program flow constraints from the flow analysis.

Program	Description	Comments
`janne_complex`	Nested loop program.	The inner loops number of iterations depends on the outer loops current iteration number.
`jfdctint`	Discrete-cosine transformation on 8x8 pixel block.	Long calculation sequences (i.e., long basic blocks), single-nested loops.
`lcdnum`	Read ten values, output half to LCD.	Loop with iteration-dependent flow.
`lms`	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.	A lot of floating point calculations.
`ludcmp`	LU decomposition algorithm.	A lot of calculations based on floating point arrays with the size of 50 elements.
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.
`minver`	Inversion of floating point matrix.	Floating value calculations in 3x3 matrix. Nested loops (3 levels).
`ndes`	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	A lot of bit manipulation, shifts, array and matrix calculations.
`ns`	Search in a multi-dimensional array.	Return from the middle of a loop nest, deep loop nesting (4 levels).
`nsichneu`	Simulate an extended Petri net.	Automatically generated code with more than 250 `if`-statements.
`qsort-exam`	Non-recursive version of quick sort algorithm.	The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.
`qurt`	Root computation of quadratic equations.	The real and imaginary parts of the solution are stored in arrays.
`recursion`	A simple example of recursive code.	Both self-recursion and mutual recursion are used.
`select`	A function to select the Nth largest number in a floating point array.	A lot of floating value array calculations, loop nesting (3 levels).
`sqrt`	Square root function implemented by Taylor series..	Simple numerical calculation.
`st`	Statistics program.	This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.
`statemate`	Automatically generated code.	Generated by the STAtechart Real-time-Code generator STARC.

Table 2 Benchmark programs (part 2)

For an embedded program or task (written in C or a similar language), the input variables can be:

- Values read from the environment using primitives such as ports or memory mapped I/O,
- Parameters to `main()` or the particular function that invokes the task, and
- Data used for keeping the state of tasks between invocations or used for task communication, such as external variables, global variables or message queues.

Therefore, we have defined multiple input values for some of the benchmarks, to be able to test and evaluate such input dependency. These inputs are provided as intervals, i.e., limits to the inputs. The inputs are stored on the web page as "input annotations" (.ann files) in SWEET format.

Loop bounds.

Each benchmark program has been run either "as is" (in single mode), or, if inputs are defined, with all inputs. The loop bounds that have been found are stored in a file at the web site. The loop bounds for the program are either exact (in the single mode case) or the maximum possible with the possible inputs, as defined on the web site. This information can be useful when doing loop bound analysis.

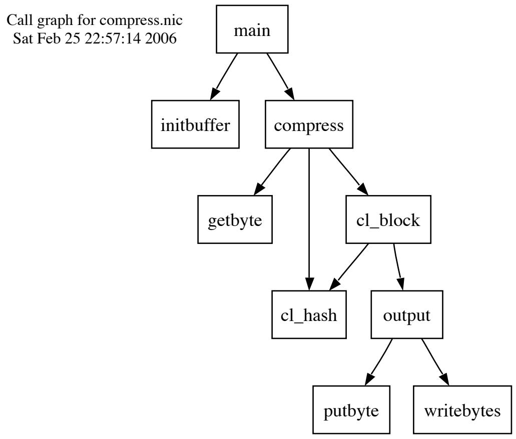


Figure 1 Example of a call graph for a benchmark program (compress).

Call graph and scope hierarchy graph.

The web site contains some graphs generated by the SWEET tool. For each benchmark file, a *call graph* (see Figure 1) is provided as a PDF file. A *scope hierarchy graph* is also available (see Figure 2), which is a context sensitive graph showing calls to functions and entries to loops. The root scope (at the top) is typically the main function, or the top function in a subgraph. The (iteration) scope is either a function or a loop, and constitutes a (possibly) looping entity in the program. The arrow from one scope to another below represents a call in the case of a function scope, or a loop invocation in the case of a loop scope. If loops are considered as a special case of (tail recursive) functions (which is a common way to look at loops), the call graph becomes the scope hierarch graph. The scope hierarchy graph is context sensitive, which means that each call site to a function creates a unique scope of the called function. The graph gives a possibility to find all, possibly looping, scopes (functions and loops) in the program.

4 Evaluation of the Mälardalen WCET Benchmarks and Ideas for Future Changes

The Mälardalen WCET benchmarks have been used extensively during their five years of existence. The benchmarks have been used mainly in two ways:

1. For evaluation of WCET algorithms and tools in research papers. The following list gives some examples of papers that have used the Mälardalen WCET benchmarks: [11, 17, 6, 16, 10].

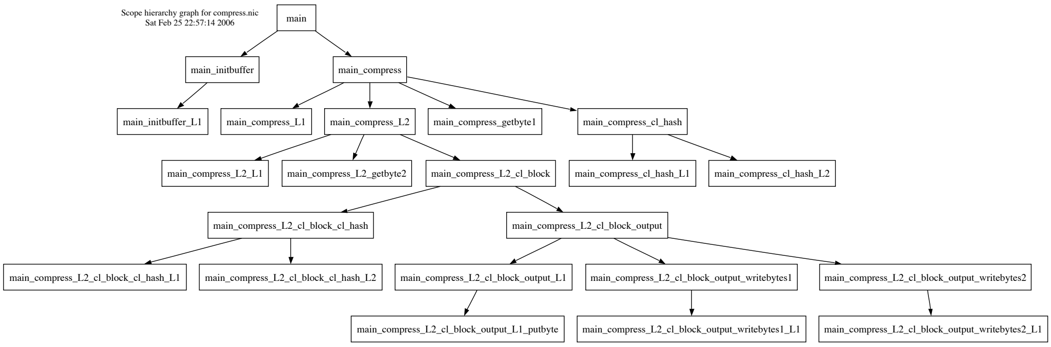

Scope hierarchy graph for compress.nic
Sat Feb 25 22:57:14 2006
main
main_initbuffer
main_compress
main_initbuffer_L1
main_compress_L1
main_compress_L2
main_compress_getbyte1
main_compress_cl_hash
main_compress_L2_L1
main_compress_L2_getbyte2
main_compress_L2_cl_block
main_compress_cl_hash_L1
main_compress_cl_hash_L2
main_compress_L2_cl_block_cl_hash
main_compress_L2_cl_block_output
main_compress_L2_cl_block_cl_hash_L1
main_compress_L2_cl_block_cl_hash_L2
main_compress_L2_cl_block_output_L1
main_compress_L2_cl_block_output_writebytes1
main_compress_L2_cl_block_output_writebytes2
main_compress_L2_cl_block_output_L1_putbyte
main_compress_L2_cl_block_output_writebytes1_L1
main_compress_L2_cl_block_output_writebytes2_L1

Figure 2 Example of a scope graph hierarchy for a benchmark program (compress).

2. For comparisons between WCET tools. A subset[1] of the Mälardalen WCET benchmarks was used during the WCET Challenge 2006 [7, 20] as the standard against which the tools were compared.

The benchmarks have been used as test programs also for other purposes, like dynamic programming [9], migration of real-time tasks [19], and scratchpad memory management [5].

During the years, we have received a lot of feedback. The issues that have been raised mainly belong to some of the categories below. We present the feedback, together with ideas for future changes.

The benchmarks are mostly small programs.

The Mälardalen WCET Benchmarks are rather small (all except two are less than 900 LOC). This can be convenient and handy. However, they typically test just a few programming constructs. The small sizes also imply that it can be hard to test how algorithms and tools scale with larger programs. Moreover, they are typically just parts of programs, i.e., they contain a rudimentary main plus some functions. Another drawback is that the whole program often fits in a cache, so it is hard to evaluate cache analyses. Therefore, it would be interesting with larger code sizes constituting full applications.

The benchmarks are not real-time industrial applications.

Many of the Mälardalen WCET benchmarks are non-real-time programs, which is acceptable if only different programming constructs need to be tested. But what often is needed is industrial real-time applications with a realistic code size, and a mix of code constructs typical for such applications. However, it seems to be hard to get such applications from the industry, and to get permission to publish the code on an open web site. One possibility is to add benchmarks that was used during the WCET Challenge 2008 (the DEBIE-1 benchmark and *rathijit_1* through *rathijit_4*). These benchmarks are available through the WCET Tool Challenge 2008 homepage [22]. We also would like to get more code examples from industry, and we have an idea how that might be done (see Section 5).

[1] The selected programs are marked with an * at the web page.

The benchmarks are mainly focussed on flow analysis.

What seems to be missing is programs that are targeting testing of program analysis for, e.g., instruction caches, data caches, branch predictions and/or other type of hardware features.

Some program constructs are missing.

Even though there are some benchmark programs containing, e.g., unstructured code and recursion, there could be more complex examples to really hard-test such troublesome constructs. Other types of program constructs that could be added is code with highly context-sensitive execution behaviour, programs with complex low-level code (like bit-operations and shifts), use of dynamic memory, mode-specific behaviour, tasks with multiple roots, tasks wrapped in a loop, and programs using function pointers.

Too few benchmarks are multi-path programs.

As mentioned above, all current benchmarks are basically single path programs. Therefore, the benchmarks should be extended to include programs with multiple input values. The possible input-value combinations should be an easily available part of the benchmark.

Weak support for measurement-based WCET analysis.

The main limitations to using the Mälardalen WCET benchmarks in end-to-end and hybrid measurement-based approaches include the following: the inputs of each program are fixed in the file and therefore different inputs cannot be supplied as parameters; bounds on the input variables are not specified, thus the turnaround time of testing is excessive; the worst-case test vector is not given and thus obtaining the actual WCET is impossible; a common set of realisitic test vectors is missing, thus different tools and techniques are very likely to generate different inputs, making comparison awkward.

Our idea to tackle these problems is to provide measurement-based versions of the benchmarks which consume a test vector from the command line. Furthermore, it is also useful to provide:

- A test harness which calls each benchmark with a predefined (large) set of test vectors. These test data will be generated *a priori* through, for example, a genetic algorithm. The rationale for such a test harness is that it provides a common framework to compare different hybrid measurement-based approaches.
- Bounds on input variables. The key part of end-to-end approaches is the test-vector generation stage, thus merely providing a static set of test vectors is not sufficient. By also supplying bounds on the input variables, therefore, allows an exhaustive exploration of the input space. These bounds are also useful for static analysis tools.

Only C programs are available.

It can be considered as a weakness that the current benchmarks only include programs written in C. After all, real-time systems are coded in many other languages, like assembler, C++, Java, and Ada. Also, more code generated from real-time systems modelling tools, like UML and Simulink/MATLAB would be interesting to add to the benchmarks.

Code for parallel systems is missing.

Benchmarks containing code for parallel systems where tasks interact might be interesting, as the WCET research moves towards multicore systems.

Precompiled binaries should be available for more types of compilers and processors.

At present, there are precompiled binaries available only for the Renesas H8300 processor (in COFF format) using gcc. It would be of interest to have precompiled binaries generated by for the most common processors in real-time systems, including SimpleScalar/M5 configuration files and compiler options. A set of often used compilers should be chosen, and the compilation should be made with a suitable number of basic flag settings.

The benchmark web site should include more results and statistics.

It would be interesting for the developers to have more results for the benchmark programs at the web site, for comparison and debugging. Also, available results for, e.g., flow analysis would let researchers concentrate on low-level analysis, and vice versa.

- Actual worst case execution time for different compilers and processors.
- The inputs that provoked the worst case execution time, and the associated path.
- The WCET estimates generated by different tools.
- Flow facts generated by different tools: loop bounds, infeasible paths, recursion depths, etc.
- Results from low-level analysis, like timing for code parts (like basic blocks), cache misses, branch prediction misses, etc.
- Statistics for the programs, like number of functions, function calls, variables, etc.

More types of graphs.

There could be more graphs for the benchmarks, e.g., control flow graphs (CFGs), and other graphs generated by the various WCET tools.

5 Conclusion and Future Work

This paper analysed the Mälardalen WCET benchmarks as they exist today. Since their introduction in 2005, they have been used extensively by WCET researchers and developers. The feedback from multiple researchers has highlighted their strengths and existing drawbacks. Taking these onboard, future work will include enhancements to the benchmarks in the following directions: better support for measurement-based analyses; larger programs to stress scalability of tools; more realistic real-time programs and a wider range of languages and code constructs to test applicability of tools. With these upcoming modifications, the WCET benchmark suite will continue to provide a suitable framework for researchers to evaluate their WCET tools and techniques in a multitude of dimensions.

We also propose a new organization of the work with the benchmarks. We need to engage the WCET community of researchers and developers be able to continuously extend the benchmarks to meet the needs of the community. Therefore, we suggest to form a committee with a responsibility for the benchmarks, and that the benchmark web site is transformed to an open wiki, with possibility for the WCET community to easily update the benchmarks and the associated meta-data.

The updates should be supervised by a committee and a steering group. The committee should be responsible for the organization of the benchmarks, the presentation of the benchmarks on the wiki, the quality check of the benchmarks, and the reporting of the state of the benchmarks to the the WCET community.

The committee and steering group should include representatives from different groups, like WCET researchers, tool vendors and real-time systems developers and industry users. The industry representatives could help in getting permission to publish real applications as benchmarks. A broad view of technical and other view should be represented, like measurement, flow analysis and low-level experts, users of small and large systems, hard and soft real-time system developers, etc.

The authors offer hosting this new web site at Mälardalen University.

Acknowledgment

This research was supported by the KK-foundation through grant 2005/0271, and the ALL-TIMES FP7 project, grant agreement no. 215068.

References

1 AbsInt Angewandte Informatik GmbH homepage, 2010. www.absint.com/ait.

2 Homepage for the SPEC CPU2006 benchmark, April 2010.
URL: http://www.spec.org/cpu2006.

3 Homepage for the Drystone benchmark, April 2010.
URL: http://www.netlib.org/benchmark/dhry-c.

4 Homepage for the EEMBC benchmark, April 2010.
URL: http://www.eembc.org.

5 Bernhard Egger, Seungkyun Kim, Choonki Jang, Jaejin Lee, Sang Lyul Min, and Heonshik Shin. Scratchpad memory management techniques for code in embedded systems without an mmu. *IEEE Transactions on Computers*, 99(PrePrints), 2009.

6 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)*, Pisa, Italy, July 2007.

7 J. Gustafsson. The worst case execution time tool challenge 2006. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Paphos, Cyprus, November 2006.

8 Homepage for the Heptane WCET analysis tool, 2006. www.irisa.fr/aces/work/heptane-demo.

9 Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 297–303. AAAI Press, 2008.

10 Raimund Kirner, editor. *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008*, volume 08003 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

11 Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1 - 3):56–67, 2007.

12 Mälardalen University. WCET project homepage, 2010.
www.mrtc.mdh.se/projects/wcet.

13 Homepage for the MiBench embedded benchmark, April 2010.
URL: http://www.eecs.umich.edu/mibench.

14 OTAWA homepage, 2010.
http://www.otawa.fr/.

15 PapaBench homepage, 2010.
http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97.

16 Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. *CoRR*, abs/0903.2251, 2009.

17 Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

18 Rapita Systems Ltd homepage, 2010.
www.rapitasystems.com.

19 Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 80–89, New York, NY, USA, 2009. ACM.

20 Lili Tan. The worst case execution time tool challenge 2006: The external test. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *Proc. 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Paphos, Cyprus, November 2006.

21 Tidorum. Bound-T tool homepage, 2010. www.bound-t.com.

22 Homepage for WCET Tool Challenge 2008 (WCC'08), 2008.
http://www.mrtc.mdh.se/projects/WCC08/.

23 Mälardalen WCET benchmarks homepage, 2010.
www.mrtc.mdh.se/projects/wcet/benchmarks.html.