15th International Workshop on Worst-Case Execution Time Analysis

WCET 2015, July 7, 2015, Lund, Sweden

Edited by

Francisco J. Cazorla

OASICS

Editors

Francisco J. Cazorla
IIIA-CSIC and
Barcelona Supercomputing Center (CAOS group)
Barcelona, Spain
`francisco.cazorla@bsc.es`

Workshop Webpage
`http://www.bsc.es/caos/wcet2015`

Sponsored by

Supported by

ACM Classification 1998
B.2.2 Performance Analysis and Design Aids – Worst-case Analysis, B.8.2 Performance Analysis and Design Aids, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, C.3 Special-Purpose and Application-Based Systems – Real-Time and Embedded Systems, C.4 Performance of Systems, D.4.7 Computers in other systems – Real time, J.7 Real time

ISBN 978-3-939897-95-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-95-8.

Publication date
July, 2015

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.2015.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

To my family in the Canary Islands and Barcelona.
Specially to David, Alex, Paula and Marcos (a natural-born fighter!).

Welcome to WCET 2015

It has been my pleasure to serve as the PC Chair of the WCET Analysis Workshop 2015 (WCET 2015). I want to start by expressing my gratitude to the authors, the PC members and the external reviewers for their effort. You have made possible to have an excellent program for WCET 2015!

WCET 2015 continues to be the reference forum for academics, practitioners and industrialists in the field of timing analysis for hard real-time systems. In this edition the WCET Analysis workshop has tried to broaden its scope to cover any aspect related to the timing analysis of computer systems. This is motivated by the fact that while in the past timing analysis has been a topic mainly for real-time systems, recently it has becoming crucial in other domains dealing with timing guarantees. This includes among other mobile computing and high-performance computing. Hence, this edition of the WCET workshop, besides papers targeting traditional WCET analysis, encouraged submissions focused on less rigorous and mature timing analysis techniques on complex multicore and manycore heterogeneous, usually COTS, architectures. For such complex architectures Execution Time Bound (ETB) estimates are derived rather than WCET estimates in the strict sense. ETB estimates are intrinsically less reliable than WCET estimates.

We received 26 good-quality submissions out of which 10 were selected for presentation and the proceedings. This has resulted in a solid program for WCET 2015 which is also strengthened with an excellent keynote by Jon Perez. Jon is the Embedded Systems research line coordinator at IK4-IKERLAN. His talk will cover aspects related to multicore, WCET and certification in the context of fail-safe mixed-criticality systems.

WCET 2015 has been also possible thanks to the financial support of the EU COST Action IC1202 on Timing Analysis on Code-Level (TACLe); the FP7 IP Project PROXIMA (`http://www.proxima-project.eu/`) and the FP7 STREP Project P-SOCRATES (`http://www.p-socrates.eu/`). Special thanks also to our supporting institution the HiPEAC Network of Excellence (`http://www.hipeac.net`).

I welcome all participants to the WCET 2015 and I encourage them to be active and enjoy the opportunity for discussion and interaction with other workshop attendees.

See you in Lund!
Francisco J. Cazorla

List of Authors

Committees

Program Chair

Francisco J. Cazorla
IIIA-CSIC and and
Barcelona Supercomputing Center (CAOS group)

Program Committee

- Benoît Triquet
  Airbus
- Björn Lisper
  University College of Mälardalen
- Christine Rochange
  IRIT
- Claire Maiza
  Grenoble INP/Verimag
- Damien Hardy
  IRISA
- Enrico Mezzetti
  University of Padua
- Franck Wartel
  Airbus
- Guillem Bernat
  Rapita Systems
- Heiko Falk
  TU Hamburg-Harburg
- Isabelle Puaut
  IRISA
- Jaume Abella
  BSC
- Luca Fossati
  European Space Agency (ESA)
- Luis Miguel Pinho
  CISTER
- Martin Schoeberl
  Technical University of Denmark (DTU)
- Peter Puschner
  TU Wien
- Tullio Vardanega
  University of Padua

External Reviewers

- Abu Naser Masud
- Andreas Gustavsson
- Catherine Vigouroux
- Daniel Hedin
- Davide Compagnin
- Hugues Cassé
- Javier Jalle
- Leonidas Kosmidis
- Linus Källberg
- Luca Pezzarossa
- Marco Ziccardi
- Milos Panic
- Mladen Slijepcevic
- Nicolas Halbwachs
- Pascal Raymond
- Rasmus Sorensen
- Torur Strom
- Wolfgang Puffitsch

A Framework to Quantify the Overestimations of Static WCET Analysis*

Hugues Cassé, Haluk Ozaktas, and Christine Rochange

University of Toulouse, France
`{casse,ozaktas,rochange}@irit.fr`

Abstract

To reduce complexity while computing an upper bound on the worst-case execution time, static WCET analysis performs over-approximations. This feeds the general feeling that static WCET estimations can be far above the real WCET. This feeling is strengthened when these estimations are compared to measured execution times: generally, it is very unlikely to capture the worst-case from observations, then the difference between the highest watermark and the proven WCET upper bound might be considerable. In this paper, we introduce a framework to quantify the possible overestimation on WCET upper bounds obtained by static analysis. The objective is to derive a lower bound on the WCET to complement the upper bound.

1998 ACM Subject Classification C.3 Computer Systems Organization – Special-Purpose and Application-Based Systems – Real-Time and Embedded Systems

Keywords and phrases static WCET analysis, uncertainty, overestimation, cache analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.1

1 Introduction

Tasks in hard real-time systems are subject to strict deadlines and any failure to meet a deadline might have severe consequences. Many approaches to hard real-time task scheduling have been proposed in the literature [4]. Based on the knowledge of the worst-case execution time (WCET) of individual tasks, they compute a task schedule that is valid in the sense that all deadlines are met.

The correctness of a task schedule relies on the confidence in the estimated WCETs of tasks. An under-estimated WCET might lead to an unsafe schedule and possibly to a violation of deadlines at runtime. Therefore, considering reliable techniques to derive those WCET estimates is crucial. The flexibility of measurement-based techniques is appealing but they cannot bring strict guarantees that the worst case has been observed or can be extrapolated. Static analysis techniques instead provide reliable estimates as soon as any detail on the hardware architecture of the target processor is known and correctly modelled[1].

Static analysis approaches determine an upper bound on the WCET based on an over-approximation of the processor state at any point in the program. For example, usual strategies to analyse the dynamic behaviour of a cache memory are based on abstract interpretation techniques that build abstract states of the cache at any program point.

By nature, WCET upper bounds are pessimistic and static WCET analysis techniques are often blamed for generating excessive overestimation. This feeling is exacerbated when

* This work was partially supported by the French research foundation (ANR) through the W-SEPT project (ANR-12-INSE-0001). See `http://wsept.inria.fr`.

[1] We are aware that this is a strong assumption that is difficult to achieve but this issue is beyond the scope of this paper.

the largest observed (measured) execution time is compared to the statically-determined WCET. The difference between the two is often exhibited as an indicator of overestimation although without any evidence (even any conviction) that the largest observed execution time was measured on the longest possible path. In the same time, it can generally not be shown that the estimated WCET value could be really reached. The goal of this paper is to show how additional numbers could be delivered together with the trustworthy WCET estimation in order to give better insight into how imprecise it might be.

Three factors can contribute to overestimating the WCET: (a) the analysis identifies a longest path that is not feasible in practice due to restrictions on input data values which are unknown (unspecified) to the analysis and/or to the incapacity of the analysis to take such restrictions correctly into account; (b) the analysis assumes a worst-case initial hardware state which cannot be observed in practice; (c) the way the analysis is performed generates overestimation (generally to reduce complexity). In this paper, we focus on factor (c) and leave the study of the two others for future work.

Our objective is to isolate the part of the estimated WCET that does not result of any over-approximation. We refer to this value as *a lower bound on the real WCET*: it represents an execution time that *is feasible* (assuming that both the sets of possible paths and initial hardware states are not over-approximated). The difference between the lower and upper bounds on the WCET gives an insight into the accuracy of the estimated WCET.

The paper is organised as follows: Section 2 introduces a framework to determine lower bounds along with upper bounds on the WCET of a task. A possible use of this framework for the cache analysis is presented in Section 3. Experimental results are reported in Section 4 and Section 5 concludes the paper.

2 A Framework to Quantify Pessimism of Static WCET Estimations

2.1 Static WCET Analysis Techniques

The usual flow for static WCET analysis is a sequence of several steps. The input is the program to analyse, more precisely its binary code required for low-level analyses. Its source code can be useful for path analyses.

In the first step (*Flow/Path Analysis*), possible execution paths are identified and encoded as a CFG (Control Flow Graph) and a set of flow facts (loop bounds, infeasible paths, etc.).

The next step (*Global Low-level Analysis*) determines the behaviour of history-based hardware schemes, such as (instruction and data) caches or branch predictors. These schemes make use of limited-capacity storage which may engender conflicts between distant parts of the program. In addition, they have been designed to improve the average execution time of the program by exploiting the execution history and hence may exhibit highly dynamic behaviours. These effects are particularly interesting to handle with static analysis approaches because time-expensive behaviours are relatively rare (but must be considered for safety reasons) and are particularly difficult to capture through measurements.

The third step (*Local Low-level Analysis*) takes the global and local behaviours of the hardware components into account to compute the possible execution times of each basic block in the CFG.

Finally, an ILP (Integer Linear Programming) system is built [10] for *WCET Computation* (IPET method). Its objective is to maximize the task execution time expressed as the sum of the basic blocks execution times weighted by their respective execution counts. These execution counts are bounded by a collection of linear constraints that express: (a) restrictions on the possible execution paths in the code, and (b) restrictions on the possible occurrences of some hardware-level behaviours.

2.2 Sources of Overestimation

The overestimation of static WCET analyses is due to three kinds of reasons:

- imprecise flow facts: in the absence of full information on possible execution paths, the WCET analysis might consider infeasible paths and those might happen to exhibit longer execution times than valid paths. The goal of flow analysis [14] is to eliminate such infeasible paths but it might fail to identify all of them for several reasons: (a) restrictions on input data values are under-specified by the user; (b) some information given by the user cannot be translated into flow facts (this depends on the annotation format considered by the WCET analysis tool, complex scenarii might be difficult to describe); (c) some flow facts derived from user annotations or automatically extracted from the source or binary code cannot be exploited during the analysis (e.g. they cannot be turned into linear constraints when the WCET is computed using the IPET method). In this paper, we assume that the applications under analysis do not suffer such issues and we consider that every relevant information of possible execution flows is known and taken into account when computing WCETs. We leave the analysis of imprecision due to incomplete information on input data for future work.
- imprecise information on the initial hardware state: the state of the processor (pipeline) and of the memory hierarchy (cache memories) has an impact on the (worst-case) execution time of a task. For example, if part of the task code already resides in the instruction cache, the overall latency of instructions fetches is shorter than if the cache is empty. In the absence of such information, the WCET analysis systematically assumes the worst-case situation and this might lead to overestimating execution times [12]. We ignore this issue in this paper and we assume that the tasks under analysis can effectively start with the worst-case hardware state.
- static analysis techniques do not unroll any possible execution path. Instead, they derive at each point in the code some properties that hold for any possible execution. This keeps the complexity of determining the longest path tractable. A consequence is that these properties might be pessimistic for a particular execution. Here, the pessimism is generated by the analysis technique and not by incomplete information on possible execution scenarii. The contribution of this paper is to quantify this overestimation which is inherent to static WCET analysis.

2.3 Proposed Framework

According to the discussion above, the execution time of an application code running on a given platform that ensures full isolation (i.e. the execution time of a task cannot be impacted by any other task or system operation) is a function of the initial hardware state and of the followed execution path:

$$\mathsf{ET} : \mathcal{H} \times \mathcal{P} \to \mathbb{N}$$

where $\mathcal{H}$ is the set of possible initial platforms states and $\mathcal{P}$ is the set of possible execution paths in the code.

The WCET of the program is defined as:

$$\mathsf{WCET} = \max_{(h,p)\in\mathcal{H}\times\mathcal{P}} ET(h,p)\,.$$

Static WCET analysis aims at determining an upper bound on the execution time. Instead of running the code with different $(h, p) \in \{\mathcal{H} \times \mathcal{P}\}$ pairs, as measurement-based approaches would do, it performs abstractions to derive, at each program point, some properties that hold for any execution path. These abstractions may lead to some undecided local behaviours which are then over-approximated to compute the final WCET estimation. For example, it can happen that the latency of an access to a cache is unknown because the cache analysis can not determine whether it will be a hit or a miss (because this depends on the execution history). One of these options must be chosen to compute the WCET. We use the term *scenario* to refer to a combination of selected options for all the unpredictable behaviors in the program and $\mathcal{C}$ denotes the set of all the possible scenarii.

The estimated (abstract) WCET can then be defined as follows:

$$\mathsf{WCET}^{\#} : \mathcal{H} \times \mathcal{P} \times \mathcal{C} \to \mathbb{N}\,.$$

Each set $\mathcal{D}$, where $\mathcal{D}$ stands for $\mathcal{H}$, $\mathcal{P}$ or $\mathcal{C}$, can be considered in several flavours:

- $\mathcal{D}^\top$ is the set of theoretically possible values.
  For example, $\mathcal{P}^\top$ is the set of possible paths expressed by the program control flow graph.
- $\mathcal{D}^+$ contains all the values that *may* (but are not guaranteed to) be feasible.
  In other words, $\mathcal{D}^+$ is a subset of $\mathcal{D}^\top$ that excludes the values that can be proven infeasible. For example, $\mathcal{P}^+$ includes all the execution paths that fulfil the flow constraints (both those computed by the flow analysis step and those specified by the user). Similarly, $\mathcal{C}^+$ is the set of all the scenarii that may occur and is considered by static WCET analysis to compute a trustworthy upper bound on the WCET.
- $\mathcal{D}^r$ is the set of all the values that *are really feasible.*
  Usually, this set is unknown or too large to be exhaustively explored.
- $\mathcal{D}^-$ is a set of values that *can be guaranteed to be feasible.*
  To illustrate this, let us consider an access to the cache that cannot be classified as a hit or a miss by the static cache analysis. If the processor is free of timing anomalies, the optimistic option is to consider the access as a hit, while the pessimistic assumption is a miss. Any scenario in $\mathcal{C}^-$ specifies a hit, and any scenario in $\mathcal{C}^+$ specifies a miss.

We have: $\mathcal{D}^- \subseteq \mathcal{D}^r \subseteq \mathcal{D}^+ \subseteq \mathcal{D}^\top$.

Usual static WCET analysis techniques produce an upper bound on the real WCET: $\mathsf{WCET}^+ : \mathcal{H}^+ \times \mathcal{P}^+ \times \mathcal{C}^+ \to \mathbb{N}$. Provided $\mathcal{H}^+$ and $\mathcal{P}^+$ are trustworthy, and assuming that the low-level analyses are correct (then $\mathcal{C}^+$ is trustworthy too), we must have $\mathsf{WCET}^r \leq \mathsf{WCET}^+$ where WCET^r is the real (but unknown) WCET of the application.

On the other hand, given that all the values in $\mathcal{H}^-$, $\mathcal{P}^-$ and $\mathcal{C}^-$ are feasible, it must be that $\mathsf{WCET}^- \leq \mathsf{WCET}^r$ with: $\mathsf{WCET}^- : \mathcal{H}^- \times \mathcal{P}^- \times \mathcal{C}^- \to \mathbb{N}$.

To summarize, $\mathsf{WCET}^- \leq \mathsf{WCET}^r \leq \mathsf{WCET}^+$. The possible overestimation on the real WCET is then quantified by $U = (\mathsf{WCET}^+ - \mathsf{WCET}^-)/\mathsf{WCET}^-$. We refer to WCET^- as the *lower bound on the WCET*[2] in contrast to WCET^+, the *upper bound on the WCET*. Only WCET^+ is a reliable estimation of the program's WCET.

Note that computing the WCET from sets of different flavours, e.g. $\mathcal{H}^- \times \mathcal{P}^- \times \mathcal{C}^+$, does not provide any useful estimate: it cannot be ordered with respect to the real WCET.

In this paper, we focus on the computation of $\mathcal{C}^-$ and $\mathcal{C}^+$ for various low-level analyses.

[2] Note that this lower bound on the WCET (WCET^-) is something different from the best-case execution time (BCET).

3 Upper-Bounding the Possible Overestimation of Low-Level Analyses

3.1 Background on Cache Analysis

Category-Based Approaches

Global low-level analyses account for the impact of history-based hardware schemes. Such schemes include instruction or data caches, or dynamic branch predictors. They offer either a fast access (a *hit* in the usual cache terminology), or a slow access (a *miss*), and are designed to maximize the number of *hits* in the average case.

A common approach to support such schemes is to assign a category to each access to the device. It has been successfully applied to LRU[3] instruction [7] and data [8] caches but also to branch prediction history tables [3] and flash memory prefetchers [5].

Instruction Cache Analysis

In [7] an analysis for LRU instruction caches introduces three categories: *Always-Hit* (AH) – on each access, the instruction block is present in the cache, resulting in a fast access; *Always-Miss* (AM) – the instruction block is never in the cache, always causing a slow access; and *Not Classified* (NC) – the behaviour cannot be predicted at analysis time. These categories have later been extended in [6, 1] with the *Persistence* category ($PERS$) that is assigned when the first occurrence of an access belonging to a loop body cannot be classified but the following accesses (in the next iterations) are *hits*.

The computation of categories is based on Abstract Interpretation (AI) techniques. A cache state is derived at each point of the program, as a function that assigns an age to each cache block: $\mathcal{S} : \mathcal{B} \to \mathcal{A}$. The age represents the position of the block in a cache set according to the LRU replacement policy: it is an integer value between 0 and A, where A is the associativity of the cache – an age equal to A means that the block is not in the cache. The *Update* function expresses the behaviour of the cache: $\mathcal{U} : \mathcal{S} \times \mathcal{B} \to \mathcal{S}$ (the accessed block gets age 0, the age of blocks younger than it is incremented by 1, the age of other blocks remains unchanged). To avoid an explosion of the number of possible cache states at a program point, abstract interpretation merges states using a *Join* function $\mathcal{J}^\# : \mathcal{S}^\# \times \mathcal{S}^\# \to \mathcal{S}^\#$ that operates on an abstract representation $\mathcal{S}^\#$ of the concrete cache states.

A category is assigned to an access to cache block $b \in \mathcal{B}$ at any program point p as follows: (i) if any state $s \in \mathcal{S}$ contains b whatever the path leading to p, the access always hits (AH); (ii) if no path leading to p produces an $s \in \mathcal{S}$ that contains b, then the access always misses (AM). Two abstract interpretation analyses are needed to draw such conclusions: (i) the MUST analysis computes the worst-case age of block b (if the worst-case age of b is less than A at point p, this means that b is always in the cache at point p) with $\mathcal{J}^\#(s_1, s_2) = s$ such that $\forall b \in \mathcal{B}, s[b] = max(s_1[b], s_2[b])$; and (ii) the MAY analysis computes the best-case age of b (if the best-case age of b is A at point p, this means that b is never in the cache at point p) with $\mathcal{J}^\#(s_1, s_2) = s$ such that $\forall b \in \mathcal{B}, s[b] = min(s_1[b], s_2[b])$.

If none of the two previous predicate holds, the access is considered as NC[4]. This category is the main source of overestimation in the WCET estimation.

[3] LRU stands for Least Recently Used and refers to the cache block replacement policy. We consider this policy in the remainder of the paper.

[4] As mentioned above, another category for blocks that remains in the cache across the iterations of a loop but cannot be classified for their first access can be assigned. This analysis is not described here for the sake of simplicity. The reader may refer to [6, 1] for further details.

Data Cache Analysis

The analysis of data caches is very similar to that for instruction caches. The additional part consists in performing a data flow analysis to discover the addresses accessed by `load` and `store` instructions and to determine which cache blocks are used [8].

While an instruction fetch addresses a single memory block (known at analysis time), an access to data performed within a loop may reference several memory blocks across iterations. The data flow analysis may find that the target of a memory access has a single value, or a set or an interval of values. A special value, $\top$, represents the cases where the analysis fails to determine the possible address values. Such a failure could reflect input-data dependent addresses or might be due to an imprecise address analysis that makes over-approximations.

The imprecision in the address analysis is also a source of imprecision in the computation of the abstract cache states used to assign the categories. When a single address value has been found for a load/store instruction, the cache state is updated in the same way as for an instruction cache. If the instruction may reference several different addresses, the analysis accounts for a possible impact on several cache sets (on all cache sets if the predicted address is $\top$. For the MUST analysis, all blocks are aged by one (whatever the number of possible address values, only one is accessed at a time); in the MAY analysis, the age of all accessed blocks is unchanged while other blocks are aged. In the end, an imprecision in the address analysis directly induces over/underestimation (MUST/ MAY) of ages in the cache states and hence on the categories: memory access with imprecise address sets are more likely to be categorized as *NC*.

Exploitation of Cache Categories to Compute the WCET

A straightforward way to account for the cost of instruction and data cache misses in the estimated WCET of a program is to add it to the expression of the execution time so that the ILP solver maximises the number of misses together with the execution time. However, this solution is reliable only if the execution time of a sequence of instructions experiencing a cache miss is less than the adding the the cost of that miss to the execution time with a hit. This is not the case when the program runs on an architecture that enables timing anomalies [11], i.e. for which the local worst case for the execution time of the sequence might be a hit.

A more reliable approach is to compute the local WCET of each basic block in the program CFG as many times as possible combinations of hits and misses for *NC* accesses exist. Then the execution count of each of these different WCET values is bounded by a combination of category-related execution counts in the ILP formulation. This approach has been implemented in OTAWA [2], the static WCET analysis tool used to carry out the experiments reported in this paper. In OTAWA, a local WCET is computed for each sequence of two basic blocks (then related to an edge in the CFG) based on contextual execution graphs [13]. It considers an optimistic or pessimistic context (availability of hardware resources) for the first and second block in the sequence, respectively, thus ensuring that an upper bound is found.

3.2 Contribution to the Lower Bound on the WCET (WCET$^-$)

Approximation in category-based instruction cache analysis clearly comes from the join operator $\mathcal{J}^\#$: even if the input cache states are precise, the result may be imprecise. In data cache analysis, another source of approximation is the address analysis that sometimes fails to derive precise address values, which leads to additional inaccuracy in the computation of

Table 1 Benchmarks.

benchmark	# accesses to I$	# loads
cjpeg_jpeg6b_wrbmp	37,328	6,187
gsm	4,2319,41	895,605
gsm_decode	1,739,854	211,854
gsm_encode	2,378,890	681,917
h264dec_ldecode_block	11,942	1,802
h264dec_ldecode_macroblock	51,373	51,373

cache states. Imprecise cache states lead to imprecise categories (*NC*, and *PERS* to a lesser extent) then to possibly overestimating WCET^+.

A simplistic approach to estimate WCET^- would consist in considering each *NC* access to the cache as a hit (similarly, each *PERS* access as a hit in the first loop iteration). However, to account for possible timing anomalies, both options must still be considered (as explained in Section 3.1). More precisely, an opposite assumption is taken for each basic block in a sequence. For example, all *NC* accesses in the first block are accounted for as misses, while those in the second block are considered as hits. This way, the local WCET of the second block is minimized (assuming no timing anomaly can occur). At the end, when all possible scenarii have been explored, the lowest local WCET is kept for each basic block.

4 Experimental Study and Discussions

4.1 Methodology

All the experiments reported here have been done using our OTAWA toolset [2] in which we implemented support for the analysis of the possible overestimation.

We considered a microprocessor architecture with a 7-stage in-order pipeline similar to that of the MPC5554 from Freescale. We assumed a single level of separated instruction and data caches with the following parameters: 4KB of size, 4-way associative with LRU replacement, with 16B cache lines and a write-allocate policy. A memory latency (for cache misses) of 20 cycles was considered. We considered an infinite-capacity write buffer, then only loads are considered in the experiments.

We have analysed several benchmarks from the MediaBench suite [9]: cjpeg_jpeg6b_wrbmp is a JPEG image bitmap writing code; gsm, gsm_decode and gsm_encode are respectively a GSM 06.10 provisional standard codec, decoder and encoder; h264dec_ldecode_block and h264dec_ldecode_macroblock are H.264 block and macroblock decoding functions. The dynamic numbers of accesses to the instruction cache and loads of these benchmarks (i.e. counts on the worst-case path) are given in Table 1.

We assume that each execution path that fulfils the flow facts specified to the WCET analysis tool and converted into constraints in the ILP formulation is feasible (i.e. no overestimation comes from the flow analysis). Similarly, we assume that any initial hardware state is feasible.

Table 2 Possible overestimation on WCET^+.

benchmark	$U = (\mathsf{WCET}^+ - \mathsf{WCET}^-)/\mathsf{WCET}^-$
cjpeg_jpeg6b_wrbmp	36.25%
gsm	161.95%
gsm_decode	96.33%
gsm_encode	216.00%
h264dec_ldecode_block	126.11%
h264dec_ldecode_macroblock	144.41%

Table 3 Possible overestimation in instruction cache analysis.

benchmark	*PERS*	*NC*	$U_{i\$}$
cjpeg_jpeg6b_wrbmp	25.0%	5.1%	0.54%
gsm	34.9%	19.6%	2.36%
gsm_decode	38.4%	29.3%	1.83%
gsm_encode	31.5%	18.7%	2.29%
h264dec_ldecode_block	52.6%	18.8%	4.84%
h264dec_ldecode_macroblock	57.7%	1.8%	0.32%

4.2 Results

Table 2 shows the possible overestimation on (WCET^+). For cjpeg_jpeg6b_wrbmp), the maximum overestimation on WCET^+ is small (36.25%), meaning that the analysis is precise in this case. The other benchmarks exhibit much larger potential overestimation, ranging from 96.33% to 216%. As we will see below, this is mainly due to the weakness of the data cache analysis we considered, which fails to compute the targets of load instructions.

In the following, we try to isolate the respective contributions of the instruction and data cache analyses to the possible overestimation of the WCET upper bound. For that purpose, we compute $U_{i\$}$ (resp. $U_{d\$}$) considering pessimistic results for the other analysis.

Table 3 shows the contribution to the overestimation by the instruction cache analysis ($U_{i\$}$). These low numbers (less than 5%) show how instruction caches can be taken into account with much accuracy in WCET analysis. There are two reasons for this. First, the instruction cache analysis generates relatively precise results: columns 2 and 3 show how many accesses are classified as *PERS* (*Persistent*, generating a hit for all loop iterations but the first one) or *NC* (*Not Classified*, then considered as a hit to be optimistic and as a miss to be pessimistic). Except for one benchmark, the number of instruction fetches that cannot be classified is less than 20%. Second, the impact of a miss on an instruction fetch can sometimes be partly hidden by stalls generated by instructions inter-dependencies, and this is captured by our pipeline analysis.

The major part of the possible overestimation on the static WCET comes from the data cache analysis, as shown in Table 4. Contrarily to the instruction cache, many accesses (from 46% to more than 80%) are categorized as *NC*. This is due to the basic address analysis implemented in OTAWA, that only handles global and stack data. Addresses of accesses to array elements are not determined by this analysis. As reported in column 4, the precise

Table 4 Possible overestimation in data cache analysis.

benchmark	*PERS*	*NC*	*unknown @*	$U_{d\$}$
cjpeg_jpeg6b_wrbmp	9.2%	46.1%	33.8%	35.29%
gsm	0.0%	55.8%	38.4%	66.25%
gsm_decode	0.0%	69.4%	52.6%	143.66%
gsm_encode	0.0%	51.7%	34.1%	190.01%
h264dec_ldecode_block	0.0%	80.5%	64.4%	82.51%
h264dec_ldecode_macroblock	0.0%	54.4%	22.1%	141.90%

target address could not be determined by the data analysis for a considerable amount of load instructions (from 22% up to 64%). An unknown target address impacts the load instruction itself, since it cannot be determined whether it hits or misses in the cache, but it also impacts subsequent loads by producing a destructing effect on the abstract cache state. Clearly, the address analysis can be identified as the weakest link here.

4.3 Possible Interpretations of the Lower Bound on the WCET

Several uses of the overestimation metric (U) – difference between the upper and lower bounds on the WCET – can be envisioned:

- An obvious application is the qualification of a particular estimated WCET. If U is small, then the estimated WCET can be considered as precise. This means that WCET^+ is not far above the real WCET (even if measured execution times look much lower). Then the estimated WCET will not drive to oversize the hardware required to meet deadlines, nor break irrelevantly the real-time constraints verification. Instead, if U is large, this may mean either that the WCET computation approach does not fit the particular profile of application under analysis, or that the way the application is coded drives too many non predictable behaviours.
- Local contributions to overestimation could also be used to implement an adaptive WCET analysis. For example, a first fast analysis could be performed with a very simple (but imprecise) method. Then the program parts exhibiting the largest overestimation could be processed again with a more precise but also more time-consuming method. Abstract interpretation is well adapted to support such an approach: to enhance accuracy, the analysis could choose which states would be merged and which would be kept separate by the *Join* operator.
- Finally, the overestimation metric could also be considered as a basis to compare different analyses with respect to the precision they can achieve.

5 Conclusion

Static WCET analysis has the reputation of overestimating WCET, which is not completely false since it is designed to compute upper bounds on the real WCET. However the size of the gap between the real and estimated WCETs is unknown (since the real WCET cannot usually be determined). Comparison of static WCET estimates to measured execution times often shows large discrepancies but nobody can tell whether the upper bound is too pessimistic or if the measured execution time was observed under particularly favourable conditions.

Our objective in this paper was to propose a framework to evaluate the possible overestimation of static WCETs. We defined a *lower bound on the real WCET* (WCET^-). This value can been seen as a complement to the traditional upper bound, WCET^+. It represents an execution time that *can* be reached, provided all information on infeasible paths and initial hardware states were known and taken into account at analysis time. The real WCET is guaranteed to be between the lower and upper bounds. We envision that these information could be useful not only to increase confidence in the static WCET analysis but also to guide code optimisations to enhance precision. It could also be used to compare analysis techniques. Note that any observed execution time that would be greater than WCET^- could be used to tighten the interval in which the real WCET falls, and thus to improve precision.

As future work, we plan to apply the proposed approach to other hardware components in single- and multi-core architectures. We will also investigate how it can be used to evaluate the precision of flow analysis techniques.

References

1 Clément Ballabriga and Hugues Cassé. Improving the first-miss computation in set-associative instruction caches. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, 2008.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, 2010.

3 Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2), 2000.

4 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

5 Niklas Holsti et al. WCET Tool Challenge 2008: report. In *Workshop on Worst-Case Execution Time Analysis*, 2008.

6 Christian Ferdinand. A fast and efficient cache persistence analysis. Technical report, Saarländische Universität (Germany), 2005.

7 Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

8 Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, 1998.

9 Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Intl symposium on Microarchitecture*. IEEE Computer Society, 1997.

10 Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modelling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium (RTSS)*, 1995.

11 Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-time systems symposium (RTSS)*, 1999.

12 Fadia Nemer, Hugues Cassé, Pascal Sainrat, and Jean-Paul Bahsoun. Inter-Task WCET computation for A-way Instruction Caches. In *Symp. on Industrial Embedded Systems (SIES)*, 2008.

13 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. 2009.

14 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference (DAC)*, 2006.

Software-enforced Interconnect Arbitration for COTS Multicores

Marco Ziccardi, Alessandro Cornaglia, Enrico Mezzetti, and Tullio Vardanega

University of Padova, Italy
{mziccard,acornagl,emezzett,tullio.vardanega}@math.unipd.it

Abstract

The advent of multicore processors complicates timing analysis owing to the need to account for the interference between cores accessing shared resources, which is not always easy to characterize in a safe and tight way. Solutions have been proposed that take two distinct but complementary directions: on the one hand, complex analysis techniques have been developed to provide safe and tight bounds to contention; on the other hand, sophisticated arbitration policies (hardware or software) have been proposed to limit or control inter-core interference. In this paper we propose a software-based TDMA-like arbitration of accesses to a shared interconnect (e.g. a bus) that prevents inter-core interference. A more flexible arbitration scheme is also proposed to reserve more bandwidth to selected cores while still avoiding contention. A proof-of-concept implementation on an AURIX TC277TU processor shows that our approach can apply to COTS processors, thus not relying on dedicated hardware arbiters, while introducing little overhead.

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

Keywords and phrases multicore, resource arbitration, interference, mixed-criticality

Digital Object Identifier 10.4230/OASIcs.WCET.2015.11

1 Introduction

An extraordinary growth in the complexity of software systems has occurred in the last decades. This complexity growth also entailed an increase in the computational demand [6] that could only be sustained by the adoption of advanced and powerful multicore and manycore systems, which have become the de-facto reference standard for computing platforms. This unrelenting transition to multicore systems also involves the application domain of real-time embedded applications (avionics, automotive, aerospace, etc.), where predictability and analyzability in the time domain are stringent requirements. The coexistence of multiple applications running in parallel on distinct cores in the same platform, however, complicates the analysis of the worst-case execution time (WCET) behavior of programs running on such systems, due to the inter-core timing interference caused by contention on access to shared hardware resources. The WCET, in turn, is the main input to schedulability analysis, which is a mandatory test for real-time systems to guarantee that tasks will meet deadlines at run time.

Notable effort has been devoted in the last few years to analyze the WCET behavior of systems deployed on modern multicore platforms. Two main research paths have been followed to cope with the inter-core interference problem [4]. One class of approaches [17, 5, 8, 14] aims at precisely analyzing the timing interference to provide safe and tight bounds, to be fed into schedulability analysis as an additive factor. However, besides the inherent complexity of deriving trustworthy bounds on complex architectures, the worst-case inter-core interference thereby computed is likely to be too large to be usable. The second class of approaches [16, 18, 21, 9] attacks this problem by carefully limiting/controlling the sources of inter-core interference in the system to make it more analyzable. This can be achieved by partitioning the shared hardware resources in a system either spatially (physical partitioning) or temporally (e.g., through a hypervisor or arbitration schemes). Resource partitioning is particularly interesting when dealing with mixed-criticality systems (MCS) [20] where the computational power guaranteed by multicore platforms allows hosting on the same target applications that are classified at different criticality levels, according to the applicable certification and qualification standard [13]. In the MCS scenario, we need to avoid high criticality tasks to be interfered by lower criticality ones in a possibly uncontrollable way.

Time and space partitioning are well-known concepts also in singlecore settings and are advocated, for example, by the ARINC-653[2] standard to provide isolation across applications running on the same core. On a multicore architecture, however, interference not only occurs between tasks running on the same core but can also arise when tasks on different cores access platform resources (e.g. shared memory, cache), often through a shared interconnect. Concurrent accesses of different criticality applications to a shared interconnect may heavily affect the performance of high criticality ones. This performance impact is often complex to quantify as it depends on the number of applications running in parallel as well as on the specific hardware arbitration policy employed. A task spending only 10% of its time fetching shared memory may suffer up to 300% interference on an eight core processor[15].

To mitigate the problem of interference and achieve analyzability, statically arbitrated interconnects are often used. Time division multiple access (TDMA) policy, amongst other, allows ruling out inter-core interference by means of temporal isolation [17]. However, the hardware support provided in Commercial Off The Shelf (COTS) multicores to arbitration policies, regulating accesses to shared interconnects, is typically oriented to achieve better average performance rather than system analyzability. Relying on specific hardware-enforced policies is, therefore, not always possible without recurring to (often unaffordable) custom hardware solutions. In this paper we propose an approach to the arbitration of hardware shared resources that relies solely on system software, without posing particular requirements on the hardware platform. Our solution applies to partitioned multicore systems where inter-core communication is regulated by the real-time operating system (RTOS). This is the case, for example, of automotive systems abiding by the OSEK/AUTOSAR[3] standards where a specific inter-core message-passing API is provided. Our approach consists in deploying a flexible software-arbitration layer through a specific API implementation, while, of course, preserving the API semantics. We exploit the inherent flexibility of software approach to enforce both a *pure* TDMA arbitration policy and a bandwidth-reservation flavour of TDMA, which is better equipped to meet mixed-criticality application requirements. We show that our technique manages to control the interference by construction and provides timing guarantees for all accesses to the interconnect. We also show that our approach is *hardware agnostic* as it can be applied to complex interconnects employing either priority-driven or round-robin policies (or a combination of the two).

This paper is organized as follows: Section 2 contextualizes our work, Section 3 introduces both the base and bandwidth-reservation versions of our software-based TDMA. An experimental evaluation is given in Section 4. Finally, Section 5 draws some conclusions.

2 Related Work

The reduction of inter-core interference on shared resources is widely recognized as an enabler of timing analysis of multicore systems. Both hardware- and software-based solutions have been considered as a means to achieve better analyzability without compromising performance. Shared interconnects, as a main source of inter-core interference, have captured the attention of the timing analysis community. From the timing analysis standpoint, inter-core interference is generally studied under the umbrella of Worst-Case Response Time (WCRT) analysis where the focus is set on using bounds on the inter-core interference in increasingly complex response time analysis equations. Analysis techniques for several arbitration schemes on COTS or custom hardware can be found in the literature, based on more or less complex models, such as arrival curves [17], event models [5] or timed automata[8]. Generally, however, the more complex the arbiter, the more pessimistic the WCRT bounds.

A classic solution to cope with contention is TDMA, where clients accessing a shared resource are served according to a fixed-time slots scheme. TDMA is widely used to manage shared interconnects on the account that it provides guaranteed bandwidth and predictable latency, regardless of the the number of contenders. Predictability, however, comes at the cost of a relative reduction in the interconnect utilization. A Round-Robin (RR) scheme can be used to improve on this aspect of TDMA: time slots are rotated only across active contenders. In comparison to TDMA, RR guarantees better utilization at the risk of being unfair to certain contenders, depending on specific access patterns [18]. An alternative approach to improve utilization consists in using a TDMA-based arbiter that varies core-to-slot allocation and slot size, and loads the *bus schedule* from memory[16]. Adaptive resource arbiters, such as FlexRay [1], have been recently studied in the context of mixed-criticality automotive software [10]. These arbiters combine static approaches as TDMA with dynamic ones, such as RR. Tough effective, hardware-based approaches are not generally implementable on COTS hardware and are inherently less flexible than their software-based counterparts.

Our approach is more related to software-based approaches, which allow abstracting away from the actual platform, and promise easy reconfigurability in response, for example, to run-time events. A software-enforced arbitration scheme for mixed-criticality systems is discussed in [21], where a memory throttling controller with variable budget assignment is exploited to limit the memory request rate of cores running non-critical tasks. This technique, however, is quite intrusive as it requires periodic checks of performance counters to monitor budget consumption. Moreover, critical tasks are only allocated to one processor and the number of supported interfering cores is limited. Our work, instead, does not pose any constraint in the assignment of critical tasks, and guarantees timing isolation regardless of the number of contenders. However, we do not account for all accesses to the shared interconnect, but only for those caused by inter-task communication. A Time-Triggered and Synchronisation-based (TTS) scheduling strategy that targets partitioned mixed-criticality systems with periodic task sets has been recently presented in [9]. TTS isolates tasks at different criticality levels and accounts for the effect of memory contention on task execution by relying on synchronization mechanisms (barriers) and fixed preemption points. Scheduling decisions are global, even though the algorithm enforced partitioning, which may incur non-negligible overheads [19]. Our approach relies on a relatively simple implementation and does not assume any specific task scheduling algorithm.

3 Software-enforced Resource Arbitration

It is not unusual for a typical embedded system RTOS to be responsible for inter-task and inter-core communication. Software specifications as, for instance, ARINC-653[2] and AUTOSAR [3] offer a communication scheme based on message-passing abstractions: two tasks are not allowed to share memory, and all communication taking place between them must be performed via messages sent through the RTOS API. The RTOS, in fact, exposes APIs for the creation of different kinds of communication channels (e.g. ports in ARINC-653 and Inter-OS Application Communication IOC buffers in AUTOSAR) and for their use, according to the respective semantics. The intuition behind our work is that, since the operating system is the mediator of all communication across applications, then it can also act as a request arbiter when such communication involves accessing a shared medium. We extend the RTOS implementation of the message-passing API to enforce a configurable TDMA arbitration policy: this is done by splitting exchanged messages into chunks and arbitrate each chunk transfer in TDMA-like time slots. TDMA frame, slots and chunk size can be configured to better meet the application requirements. To address specific needs of mixed-criticality systems, a more *flexible* TDMA-based arbitration is also proposed that allows reserving more than one communication slot to selected cores. This version of TDMA enables the provision of better bandwidth levels to cores running high-criticality applications.

The so-enforced software-based TDMA arbitration mechanism (SW-TDMA) has several advantages. First of all, the combination of message-passing API and TDMA enforces isolation between tasks running on different cores and makes the effects of bus contention completely predictable. Moreover, the adoption of software-based arbitrations is transparent to user applications as the arbitration mechanism is implemented at kernel level and does not affect either the syntax or the semantics of message passing APIs. Finally, the ability to reserve more than one arbitration slot to selected cores enables more complex arbitration policies to be designed without the need for expensive hardware. As an improvement over system flexibility, bandwidth levels can also change at run time in reaction to specific events or changes in the mode of operation. In the following we present our approach to enforce flexible TDMA-based arbitration schemes for inter-core communication on a shared interconnect. Before entering into details, we introduce our model and notation.

3.1 Assumptions and Model

We consider a multicore processor architecture with n cores $\{C_0, ..., C_{n-1}\}$, each one equipped with its own local memory (e.g. cache and/or scratchpad). Cores also share a global memory that can be accessed via a simple system bus or more complex interconnects as, for instance, crossbars. We assume that all cores share a common time source TS. We consider partitioned systems, where each application is statically assigned to one of the n cores. An application allocated to core C_i comprises a set of tasks (schedulable on C_i) and is granted exclusive use of a local memory area, separated from other applications, for storing its code and data.

Communication across tasks is only allowed through the RTOS, which becomes responsible for all memory transfers. To that extent, the RTOS API exposes a pair of procedures to read/write data from/to a communication channel shared across tasks:

(i) *receive_message*: copies a message (sequence of bytes) from a communication channel to the calling task's memory;

(ii) *send_message*: copies a message (sequence of bytes) from the task's memory to a communication channel.

Inter-task communication between tasks running on the same core C_i does not follow this scheme as message channels are stored directly in C_i's local memory. We assume the RTOS is able to distinguish between local and global communications in order to avoid unnecessary arbitrations. It is worth noting that these assumptions on the software stack are general enough to allow our technique to be used underneath the inter-task communication mechanisms provided by specifications such as ARINC-653 and AUTOSAR.

In the following we adopt the standard TDMA naming convention. On a TDMA bus each of the n contenders is assigned a *time slot*. We use the symbol ss to indicate the

Table 1 Notation.

fs	Size of a frame in clock cycles with respect to TS
ss	Size of a slot in clock cycles with respect to TS
cs	Size of a chunk: amount of bytes transferred in one slot
ms	Size of a message in bytes
$fb(t_i)$	Start time of the current frame at time t_i
$sb(C_i, t_i)$	Start time of the first slot assigned to core C_i following t_i

duration of a slot in clock cycles. A set of n time slots is called a *frame*. Similarly to slots, we use the symbol fs to indicate the duration of a frame in clock cycles. Inside a frame each contender is granted a dedicated slot to access the interconnect and transfer data: a *chunk* is the largest portion of memory that a task can transfer to/from global memory inside a TDMA slot when run in isolation. We call cs the size of a chunk expressed in bytes. Since a conveyed message may consist of several chunks we use ms to address the message size in bytes. Accordingly, $fb(t_i)$ refers to the start time of the current frame at time t_i (with respect to TS) and $sb(C_i, t_i)$ refers to the start time of the first available slot to core C_i after time t_i. Table 1 summarizes the notation used in the paper.

3.2 Software-enforced TDMA

The actual operation and efficiency of a SW-TDMA scheme depend on how *frame*, *slot* and *chunk* size, as defined in the previous section, are configured. We start from defining the size of a chunk as it affects the other two parameters. On hardware implementations of TDMA, the value cs and the size of a slot in clock cycles (ss) are part of platform's design choices along with other hardware characteristics (e.g. the bus operating frequency). Under these circumstances, the size of a slot is subject only to physical constraints and is designed so that the maximum bandwidth is achieved: the smaller possible size that permits the transmission of exactly cs bytes. On SW-TDMA, however, other sources of overhead have to be taken into account. The value ss, expressed in clock cycles with respect to the shared time source TS, must in fact consider (i) the cost of transferring cs bytes to global memory (with no contenders), and (ii) the cost of arbitrating the request. The exact value of ss can be either computed via static analysis or derived by means of extensive measurements. Once ss is known, the size of a frame fs is straightforwardly computed as $n \cdot ss$. As in standard TDMA, each core is allocated one slot inside a frame. We call $slot[C_i]$ the slot allocated to core C_i. As TDMA divides time into frames, in order to select the slot in which each core is allowed to transfer, the RTOS must be able to identify, at each time t_i, the start time of the *current frame*. At every time t_i the current frame as the frame starting at time $fb(t_i)$, where $fb(t_i)$ is such that $t_i \geq fb(t_i)$ and $t_i < fb(t_i) + fs$. Equation 1 can be used to compute $fb(t_i)$.

$$fb(t_i) = fs \cdot (\lfloor t_i/fs \rfloor) \qquad (1)$$

It is worth noting that if the size of a frame is a power of 2 ($fs = 2^{bits}$) the above operation has a blazingly fast implementation using bit-shifts: $fb(t_i) = (t_i \gg bits) \ll bits$.

Consider now an inter-core communication request issued by task τ_j running on core C_i (either a *receive_message* or *send_message*). Since a core is allowed to transfer data only inside its slot in the frame and since each slot can only hold cs bytes, the RTOS divides the data into $\lceil ms/cs \rceil$ chunks of up to cs bytes and operates each chunk transfer separately (see Figure 1). When a chunk transfer is issued on core C_i, its issuing time t_{req} is captured and the current frame's start time fb is derived using Equation 1. To correctly mimic TDMA,

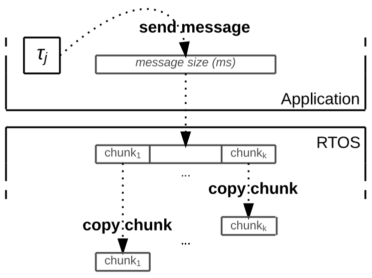
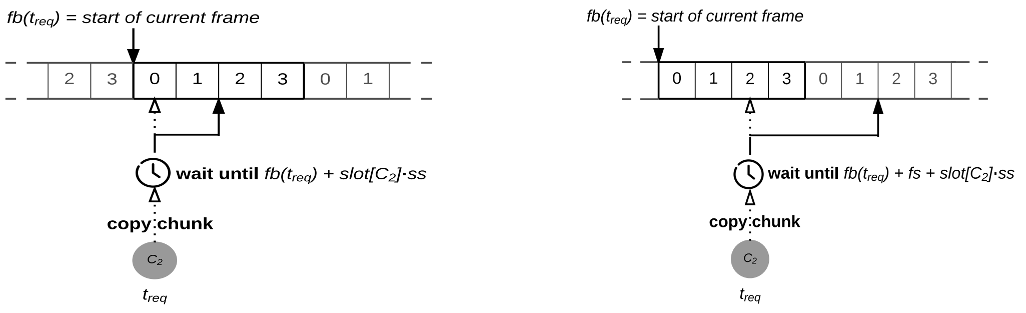


Figure 1 At the RTOS level, a message is split into $k = \lceil ms/cs \rceil - 1$ chunks of size cs and one chunk of size $ms - k \cdot cs$. Each chunk transfer will take place inside a TDMA slot assigned to C_i.


fb(treq) = start of current frame
2 3 0 1 2 3 0 1
wait until fb(treq) + slot[C2]·ss
copy chunk
C2
treq
fb(treq) = start of current frame
0 1 2 3 0 1 2 3
wait until fb(treq) + fs + slot[C2]·ss
copy chunk
C2
treq

(a) Request fit inside current frame. **(b)** Request shifted to following frame.

Figure 2 SW-TDMA arbitration examples where a request either (a) arrives before its slot – and served in the current frame – or (b) misses the corresponding slot in the current frame.

the requested transfer is deferred until the beginning of the next slot assigned to core C_i. We refer to such time as $sb(C_i, t_{req})$ and we define it as:

$$sb(C_i, t_{req}) = \begin{cases} fb(t_{req}) + slot[C_i] \cdot ss & \textbf{if } t_{req} \le fb(t_{req}) + slot[C_i] \cdot ss \\ fb(t_{req}) + fs + slot[C_i] \cdot ss & \textbf{otherwise} \end{cases} \tag{2}$$

If the request arrived before the beginning of the corresponding core's slot inside the current frame then $sb(C_i, t_{req})$ is set to such value. Otherwise, the request missed its slot in the current frame and has therefore to wait until its slot in the next frame. The meaning of Equation 2 is graphically represented in Figure 2. Two alternative approaches are possible to intercept time $sb(C_i, t_{req})$: we may either *poll* on a cycle counter until it reaches $sb(C_i, t_{req})$, or use a *timer interrupt*. Polling is simple to implement but requires a time source to be locally available to each core (no interference in accessing it). The alarm-based solution avoids polling and works well even if no core-local time source is available but introduces additional delays due to interrupt handling.

3.3 Bandwidth Reservation TDMA

A more flexible arbitration scheme can be defined on top of the baseline SW-TDMA arbitration, to provide higher bandwidth guarantees to a selected subset of cores. When standard TDMA is used, a frame is divided into n slots, where n is the number of contenders (cores, in our SW-TDMA): to allocate more bandwidth to some of the cores a frame is divided into m slots, with $m > n$. Once each core is associated to a slot, the remaining $m - n$ slots can be used to reserve more bandwidth to selected contenders. Let us call $core[j]$ the core that has been allocated to the j-th slot, with $j \in [0, m - 1]$.

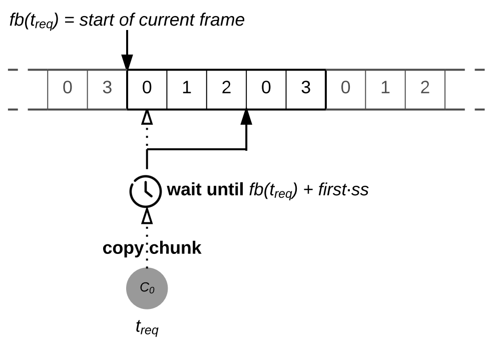

fb(treq) = start of current frame
0
3
0
1
2
0
3
0
1
2
wait until fb(treq) + first·ss
copy chunk
C0
treq

Figure 3 A chunk transfer request is assigned to the first available slot for core C_0.

Once a chunk request is issued on core C_i, the request time t_{req} is saved and the start time of the current frame, $fb(t_{req})$, is computed following exactly Equation 1. Then, the starting time $sb(C_i, t_{req})$ of the first slot at which core C_i is allowed to transfer data must be selected similarly to the basic TDMA case, with the only difference that more than one slot may be allocated to the same core within a given frame. This difference is reported in Equation 3 where the term $first$ stands for the first slot granted to core C_i.

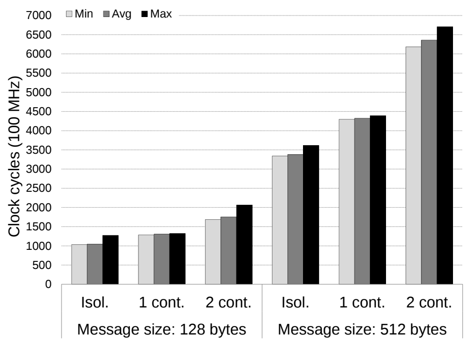
$$sb(C_i, t_{req}) = \begin{cases} fb(t_{req}) + first \cdot ss & \textbf{if } \exists first \mid first = \min\limits_{j \in [0,m-1]} \left\{ \substack{core[j]=C_i \ \wedge \\ t_{req} \leq fb(t_{req}) + j \cdot ss} \right\} \\ fb(t_{req}) + fs + first \cdot ss & \textbf{otherwise } (first = \min\limits_{j \in [0,m-1]} core[j] = C_i) \end{cases} \quad (3)$$

Figure 3 illustrates the first case of Equation 3: a slot $first$ allocated to C_i exists in the current frame and its start time follows t_{req}. Otherwise, the message transfer is deferred to the C_i's first allocated slot in the next frame.

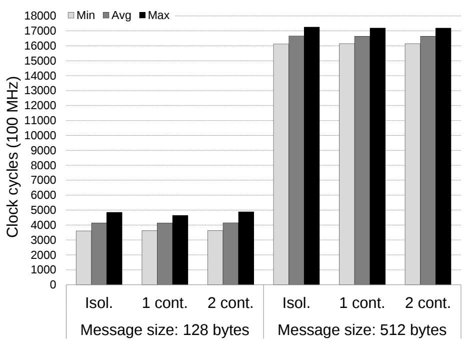
It is worth noting that the bandwidth-reservation SW-TDMA scheme can also support run-time changes of the slot allocation policy. The RTOS, in fact, might decide to do so in reaction to certain events or changes in the operational mode.

4 Proof-of-Concept Evaluation

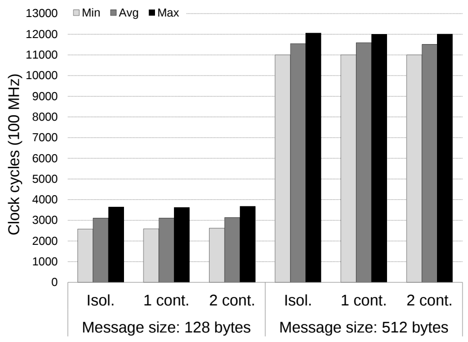
We now evaluate how well our software-based arbitration scheme is able to enforce TDMA arbitration via software on top of interconnects employing different hardware arbitrations (e.g. round-robin). We want to demonstrate that our SW-TDMA suffers no variable interference depending on the number of contenders. Finally, we want to prove that bandwidth reservation TDMA can be used to provide different bandwidth guarantees to different cores. We developed a proof of concept implementation of our approach on a realistic scenario within the automotive application domain, where silicon vendors already provide multicore solutions [11] and the AUTOSAR standard advocates partitioned multicore RTOS[3]. We run our experiments on an Infineon AURIX TC277TU [12] equipped with three TriCore CPUs. Each core has local data and instruction scratchpads. The TC277TU comes with a Shared Resource Interconnection (SRI), a crossbar that connects all cores to the Local Memory Unit (LMU) and to the Program Memory Unit (PMU). The LMU controls a shared 32KB SRAM while the PMU controls three banks of flash memory. The SRI crossbar implements both priority-driven and round robin arbitrations. Priority levels from 0 to 7 are provided. Only levels 2 and 5 can be assigned to more than one core, arbitrated according to standard round-robin policy. In our experimental setup all cores are in the same round-robin group. In addition, the crossbar employs separate hardware channels for each slave resource connected to it. That is, different SW-TDMA arbitrations can be implemented on each crossbar's slave.



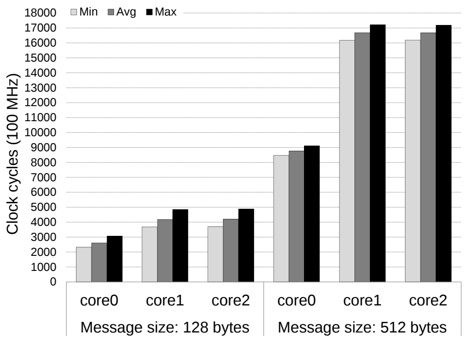
(a) Standard SRI hardware arbitration. All cores in the same round-robin group.



(b) SW-TDMA with 1024-cycles frames and 32-bytes chunks.



(c) SW-TDMA with 1024-cycles frames and 48-bytes chunks.



(d) Bandwidth-reservation SW-TDMA with 1024-cycles frames of 4 slots (2 allocated to CPU0) and 32-bytes chunks.

Figure 4 Execution times in clock cycles (100MHz) for message send primitives with messages of size 128 and 512 bytes. Figures 4a, 4b and 4c report the cost for sending a message in isolation or with one or two contenders respectively; min and max are the absolute values observed on any of the cores. In Figure 4d all cores run a task sending messages, min and max values are per-core.

Our SW-TDMA arbitration has been implemented in ERIKA Enterprise [7], an OSEK/VDX compliant RTOS. Test applications are developed on top of ERIKA and make use of a user-level library implementing the AUTOSAR IOC communication layer. Tasks' data and instructions are placed on cores scratchpads while IOC buffers are located inside the shared SRAM. Test applications are designed to measure the cost of sending messages of different sizes under different workloads: a single task runs on each core and all three tasks send messages of the same size to different buffers (no mutual exclusion across tasks is required). Tasks share the same period and are synchronized through a barrier so that to produce maximum interference. Measurements are collected via the on-core cycle counter running at 100MHz and are stored in the scratchpad (low latency tracing with no contention). As a first experiment, we evaluated the cost of accessing the SRAM through the crossbar *as-is*, with no modification. Results observed for sending messages of 128 and 512 bytes are reported in Figure 4a. The cost of sending a message evidently grows with the number of contenders. We witnessed an increase of 27% and 22% in the maximum observed execution time (MOET) when two tasks try sending a message (of 128 and 512 bytes respectively) at the same time, with respect to performing the same activity in isolation. Adding one more contender causes the MOET to become 93% and 86% bigger than in isolation, for messages of 128 and 512 bytes. This non-linear growth of interference can be explained by the very structure of the SRI. The AURIX crossbar, in fact, divides a transaction into two phases:

(i) a *request phase* where the address is sent and the request is arbitrated, and
(ii) a *data phase* where actual data is transmitted.

No more than one request can be arbitrated at a time, however, data and request phases of different transactions can be pipelined. This also explains why the MOET is only $\sim 90\%$ higher when all the cores access the crossbar. Different access patters, however, may cause more interference to occur.

In Figure 4b we report the execution times observed when sending messages of 128 and 512 bytes under SW-TDMA, with frames of 1024 clock cycles and chunks of 32 bytes. The graph shows that the cost for sending a message does not depend any more on the number of contenders, which is exactly what we were after. It is worth understanding which portion of a slot is wasted to take software arbitration decisions. Under the above configuration a slot is approximately 342 clock cycles. By means of extensive measurements, we observed that a core in isolation is capable of transferring up to 57 bytes in 342 clock cycles. Therefore, if software arbitration with chunks of 32 bytes is employed, the throughput of a core is reduced by a 44%. In Figure 4c, however, we show that throughput can be enhanced while maintaining isolation across cores. The graph reports a different configuration in which frames are still of 1024 clock cycles but chunks are now of 48 bytes. The cost of sending a message is still constant irrespective of the number of contenders while the throughput is increased and execution times decreased. Under this configuration SW-TDMA reduces the throughput of a core in a slot only by a 16%. As expected, when comparing our approach to the original SRI crossbar we notice that the average cost of sending a message is increased by almost 80%, for messages of 128 bytes. The magnitude of this result is explained by the fact that the AURIX SRI crossbar also allows some degree of pipelining. A fair comparison would require to evaluate our SW-TDMA against pure TDMA or RR interconnects with no splitting. In particular, the results obtained for SW-TDMA are extremely near to the theoretical performance offered by hardware TDMA. From the mixed-criticality standpoint, SW-TDMA caters for isolation among criticality levels, regardless of bus access patterns.

The results we observed on bandwidth-reservation TDMA are shown in Figure 4d. The frame is again set to 1024 cycles but divided into 4 slots for chunks of 32 bytes. Slot 0 and slot 2 are both allocated to core 0. Results show that, as expected, sending a message on core 1 and core 2 costs exactly as in Figure 4b where a frame of 1024 was divided into 3 slots each for a chunk of 32 bytes. Core 0 instead executes twice as fast, as it is granted two slots. Our evaluation highlights that bandwidth-reservation SW-TDMA allows assigning larger portions of the shared interconnect to a selection of cores while still preserving isolation, in keeping with mixed-criticality requirements.

5 Conclusion

In this paper we propose a solution to enforce isolation among cores accessing shared resources that does not rely on the availability of specific hardware. SW-TDMA is particularly interesting in the context of mixed-criticality applications where critical tasks must not be interfered by lower criticality tasks. We provided experimental evidence that SW-TDMA ensures isolation while introducing only a 16% throughput drop due to software arbitration. A more flexible scheme, named bandwidth reservation TDMA, has been also introduced and evaluated. The latter technique provides selected cores with more bandwidth without affecting isolation and is much suited to guarantee a better quality of service to critical tasks. As future work we plan to investigate more complex schemes as, for instance, random permutation. We also plan to further reduce the overhead introduced by software arbitration.

Acknowledgments. The work presented in this paper has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA, www.proxima-project.eu).

References

1 ISO 17458-1. Road vehicles – FlexRay Communications System – Part 1, 2013.

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

3 AUTOSAR. AUTOSAR Release 4.1. `http://www.autosar.org/`, 2014.

4 A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2015.

5 D. Dasari et al. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Proc. of IEEE 10th Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

6 R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.

7 Evidence. ERIKA Enterprise. `http://erika.tuxfamily.org/`, 2015.

8 G. Giannopoulou et al. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proc. of ACM International Conference on Embedded Software*, 2012.

9 G. Giannopoulou et al. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proc. of 11th ACM Intl. Conference on Embedded Software*, 2013.

10 D. Goswami et al. Time-triggered implementations of mixed-criticality automotive software. In *Proc. of 13th Design, Automation & Test in Europe Conference*, 2012.

11 Infineon Technologies AG. AURIX™ TriCore™. `http://www.infineon.com/aurix`, 2012.

12 Infineon Technologies AG. TC27x Manual. `http://www.infineon.com/aurix`, 2014.

13 International Electrotechnical Comission. *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0*, 2009.

14 T. Kelter et al. Evaluation of resource arbitration methods for multi-core real-time systems. In *13th International Workshop on Worst-Case Execution Time Analysis*, 2013.

15 R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *Proc. of 13th Conference on Design, Automation and Test in Europe*, 2010.

16 J. Rosen et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of 28th IEEE Real-Time Systems Symposium*, 2007.

17 A. Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium*, 2011.

18 H. Shah et al. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Proc. of Design, Automation & Test in Europe Conference*, 2011.

19 L. Sigrist et al. Mixed-criticality runtime mechanisms and evaluation on multicores. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

20 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of 28th IEEE Real-Time Systems Symposium*, 2007.

21 H. Yun et al. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proc. of 24th Euromicro Conference on Real-Time Systems ECRTS*, 2012.

Timing Analysis of Event-Driven Programs with Directed Testing

Mahdi Eslamimehr and Hesam Samimi

Communications Design Group, SAP Labs, Los Angeles, USA
`{eslamimehr,hesam@ucla}@ucla.edu`

Abstract

Accurately estimating the *worst-case execution time* (WCET) of real-time event-driven software is crucial. For example, NASA's study of unintended acceleration in Toyota vehicles highlights poor support in timing analysis for event-driven code, which could put human life in danger. WCET occurs during the longest possible execution path in a program. Static analysis produces safe but overestimated measurements. Dynamic analysis, on other hand, measures actual execution times of code under a test suite. Its performance depends on the branch coverage, which itself is sensitive to scheduling of events. Thus dynamic analysis often underestimates the WCET. We present a new dynamic approach called *event-driven directed testing.* Our approach combines aspects of prior random-testing techniques devised for event-driven code with the directed testing method applied to sequential code. The aim is to come up with complex event sequences and choices of parameters for individual events that might result in execution times closer to the true WCET. Our experiments show that, compared to random testing, genetic algorithms, and traditional directed testing, we achieve significantly better branch coverage and longer WCET.

1998 ACM Subject Classification B.2.2 Performance Analysis and Design Aids, D.2.5 Testing and Debugging

Keywords and phrases worst-case execution time, timing analysis, event-driven, directed testing

Digital Object Identifier 10.4230/OASIcs.WCET.2015.21

1 Introduction

Real-time event-driven systems have become ubiquitous, from high performance servers to smart devices. The correctness of such systems becomes of utmost importance when human safety is concerned. Testing and analyzing real-time event-driven programs is notoriously hard, mainly due to the non-linear control flow in the execution of event handlers. Dependencies are complicated to track in event-driven code, leading to subtle bugs that can go unnoticed with traditional event-driven testing. For example, since 2002 more than 89 people have been killed and 60 injured, due to the unintended acceleration of Toyota cars, which has made the corporation recall more than 1 million cars due to safety issues. The 2011 NASA study of unintended acceleration in Toyota vehicles [12] highlights poor support in timing analysis for event driven code as a contributor to the safety holes.

Thus a precise calculation of the *worst-case execution time* (WCET) of real-time event-driven software is crucial. WCET is a much studied problem for event-driven software. For time-critical embedded systems, designers must avoid excessive over-provisioning of task deadlines, because it wastes processor's availability that could otherwise be used for other functions. Designers shall also avoid under-provisioning as it can undermine the validity of the computation or lead to partial or complete loss of functionality.

The exact WCET in a program happens during the longest possible execution path among all processes.[1] In an event-driven code events can be fired at arbitrary times and a scheduler may interrupt the execution of one event handler to yield to another new or unfinished event handler. The choice of scheduling for the execution of handler codes affects the executed paths, since there can be shared state and exclusivity requirements among events. Thus WCET can only be precisely computed by checking the execution times of all possible execution paths over every possible schedule for the execution of every possible combination of triggered event handlers.

Static analysis can find an upper-bound on the true WCET [18] without executing the program, yet it is necessarily conservative and can be difficult or currently impossible to perform on arbitrary code. Dynamic approaches [18], on the other hand, are easier to perform since they measure a program's WCET by executing it on a test suite and tracking the longest execution time. Thus the accuracy of dynamic WCET calculation depends on the percentage of all possible execution paths covered. A straightforward way to estimate a program's WCET is to sum up the WCET calculations of individual handlers when executed in isolation. Yet this estimate will necessarily be conservative and an upper-bound for the true WCET. This is because the WCET of one event may not occur on conditions that would cause the WCET of another event due to intra-dependencies among event handlers. The tested WCET is a lower-bound on the true WCET that occurs during some run of the program. In slogan form:

$$\textit{tested WCET} \leq \textit{true WCET} \leq \textit{static WCET}$$

Therefore the success of a dynamic approach is closely tied to the branch coverage of the test suite used. Building a test suite with high coverage is a known challenge, and is even harder for event-driven software. Not only a tester must choose appropriate inputs to guide the execution to unexplored paths, she must devise a suite of *event sequences* (schedule of events fired) that dictate the execution context switches between concurrent processes. For creating an event sequence, a tester must decide on the number of events, the types of events, the argument values for each event, as well as a concrete timing schedule for the interrupts.

Previous work on testing event-driven software uses event sequences that are generated randomly [2] or by genetic algorithms [1]. In the domain of sequential software, the idea of *directed testing* with *concolic execution* (hybrid concrete and symbolic) has been receiving much attention in both research and practice. The idea is to execute the code concretely to explore one possible execution path, yet simultaneously employ symbolic execution to collect path constraints from each condition on the control-flow. Those constraints will be modified and solved iteratively, in order to find input values for subsequent runs, aiming to force the execution to unseen paths and increase the coverage. However, we showed previously [4, 5] that using the classical directed testing on concurrent and event-driven programming paradigms is not as successful, because the scheduling of concurrent processes is a factor.

The challenge. Improve the accuracy of WCET measurement of dynamic test-based approaches by devising a method that achieves higher branch coverages.

Our approach. We present a new technique called *event-based directed testing* for testing of event-driven software. Our technique combines good features of random testing of event-driven software with the idea of directed testing for sequential code. Similar to directed

[1] ignoring per-statement execution time which depends on the underlying machine architecture.

testing, after each round of concolic execution we generate a new scenario for further testing. Unlike directed testing where the scenario is uniquely described by input values, here we need to generate an event sequence. An event sequence not only provides input values for individual events, it provides a scheduling for the set of events that execute concurrently.

We have implemented our technique in an existing tool called VICE [4]. This approach, implemented by VICE, was previously used for maximum stack size analysis, and we have augmented it to automatically test event-driven software without a human in the loop. Our experiments show that compared to random testing, genetic algorithms, and traditional directed testing, we achieve significantly better branch coverage, and subsequently longer WCET. For 8 out of 11 benchmarks, we achieve more than 70% branch coverage. Compare to random testing, genetic algorithm, and traditional directed testing we improve WCET by 203%, 176% , and 97%, respectively.

The rest of this paper. In the next section we illustrate our approach via an example. in Sec. 3 we formalize our approach, and in Sec. 4 we show our experimental results.

2 Example

We now explain our approach through an example program shown in Listing 1—a simplified version of our antenna benchmark. There are two event handlers: `main` and `alt`. The program has three if-statements, hence six branches. The aim is to estimate the WCET for the entire program, represented by the `main` event, under a bounded number of possible interrupts.

An event sequence of a bounded length (here we pick four) is used to impose a particular scheduling of interrupts that occur during the execution of `main`. Each event in the sequence is a 4-tuple $\langle id, name, args, timeout\rangle$. Each named event must be fired with the given arguments and maximum allotted time specified by the timeout value. If the identifier is seen before, this means rather to resume the execution of an earlier interrupt. Whether the event handler is finished or interrupted by the scheduler due to a timeout, the scheduler proceeds to fire the next event in the sequence. Once the sequence is finished, the scheduler is left alone to let all unfinished interrupts run their course one by one. Our objective is to produce new sequences that will lead us to new paths in search of longer execution times.

Testing proceeds in rounds. In each round we choose a new event sequence to execute. For each tuple, we pick randomly either a new or old identifier, and, in the former case, the name of the event. We use concolic execution to determine the arguments. It is possible to generate invalid sequences, since the *interrupt mask register* follows interrupt rules (e.g., a handler cannot interrupt itself), in which case we move onto the next round. During the execution of each round, we monitor the branch coverage so far, as well the execution time of the `main` interrupt, which will be used to determine the WCET thus far.

Round one. In the first round, the arguments too are chosen randomly so we might begin with this event sequence (we omit the timeouts and show names and ids together):

$$[\langle \texttt{main}, (723452)\rangle,\ \langle \texttt{alt1}, (-10038)\rangle,\ \langle \texttt{main}, \_\rangle,\ \langle \texttt{alt1}, \_\rangle]$$

The concolic execution will fire the first event. `main`'s execution continues with calling `dispatch_data` in line 4 and we collect the constraint $data_1 = msg$, which maps the actual parameter (line 4) to the formal (line 6). We use $data_1$ to represent the symbolic value of `data_1`, and likewise for msg and `msg`. `main`'s execution reaches the second if-statement of line 9. Assuming only arithmetic equations from conditionals are collected as constraints,

Listing 1 Example program.

```
 1 program Sample { entrypoint main = antenna.main, alt = antenna.alt; }
 2 component test_antenna {
 3    field sending:bool = false;
 4    method main(data_1:int):void { dispatch_data(data_1); // code... }
 5    method alt(data_2:int):void { dispatch_data(data_2); }
 6    method dispatch_data(msg:int):void {
 7      local res:int, tmp:int = random(100)
 8      if (sending) return; else sending = true;
 9      if (-2048 < msg && msg < 1024) res = check_and_send(msg,tmp);
10      sending = false;
11      return;
12    }
13    method check_and_send(s:int, t:int):int {
14      if (s==512)
15          return 1;
16      // send...
17      return 0;
18    }
19 }
```

we proceed to collect $-2048 < msg \wedge msg < 1024$ at line 9. Since `msg` 's concrete value is 723452, the conditional evaluates to `false`. Let us assume this handler gets interrupted before resetting the `sending` flag. `alt1` is launched now, which proceeds to calling `dispatch_data`, and the constraint $data_2 = msg$ is collected. The `alt1` event terminates early at line 8, due the `sending` flag being set. `main` now resumes and runs to completion. The fourth event in the sequence is skipped since `alt1` is already terminated. During the first round, the path (of statements visited in all handlers) and the elapsed time for the execution are recorded. The branch coverage was 50% (two false branches and one true branch out of six). The execution never reached any farther than line 12.

After the completion of the first round, a new sequence of event ids and names is generated randomly. Each event in the sequence will be paired with argument values obtained by solving for all three constraints that we collected above. For example, for the next round we may produce the event sequence:

$$[\langle \texttt{main}, (-338)\rangle, \langle \texttt{alt1}, (1001)\rangle, \langle \texttt{alt2}, (6)\rangle, \langle \texttt{main}, \_\rangle]$$

where the solution set for the first element was $data_1 = data_2 = msg = -338$, and so on.

Round two. This time `main`'s execution takes the true branch in line 9 and goes on to invoke the `check_and_send` routine. Consequently we collect new constraints $msg = s \wedge tmp = t$. Let's assume at this point `main`'s handler times out. The second and third handlers—`alt1` and `alt2`—will run and terminate early one at a time, again due to boolean flag. The last event in the sequence is `main`, which resumes in the body of `check_and_send`. It encounters a new constraint at line 14: $s = 512$, which evaluates to `false` on the current argument value. The handler runs more code at line 16 and finishes. At this round, a longer execution is found due to a better branch coverage, when the handler's execution reached into deeper parts of the program. The total branch coverage over the first two rounds is 83% (5 / 6).

Round three. Suppose in the third round we use the same sequence as before, yet timeouts are set differently, so that both `main` and `alt1` handlers proceed to invoke the routine and terminate at line 12. We note that while branch coverage wasn't improved, a longer WCET so far was measured. Upon the completion of the execution of this round all collected constraints are sent to the constraint solver to generate arguments for each event in a new sequence, getting the solution $data_1 = data_2 = msg = s = 512$. Thus, the new event sequence for the next round might be:

$$[\langle \texttt{main}, (512)\rangle, \langle \texttt{alt1}, (512)\rangle, \langle \texttt{main}, \_\rangle, \langle \texttt{alt1}, \_\rangle]$$

Round four. In the fourth round, let us assume `main`'s timeout is small and it quickly gets interrupted by `alt1`, after executing the first statement. `alt1`'s handlers proceeds to calling `check_and_send` and this time will take the true branch of the if-statement. Again, before resetting the `sending` flag, the execution may yield back to `main` which will terminate early at line 8. During this round, we achieved 100% coverage, yet we observe that the longest execution hence the largest WCET so far occurred during the previous round.

More rounds. New rounds will be carried out until we measure no improvement in both WCET and the branch coverage. At this point, the algorithm terminates.

Discussion. The example illustrates several strengths of our approach. First, the combined monitoring of WCET and branch coverage as the terminating condition for the algorithm leads to a more accurate estimate of WCET, as opposed to relying on one of them only. Second, the combination of a randomly chosen sequence of interrupts, with arguments for each obtained by constraint solving leads to the exploration of a diverse set of control-flow paths. For example, the chance of reaching line 15 with input sequences generated randomly or by genetic algorithms is very small. Third, we get good branch coverage with a fairly short number of event sequences, each with a small length, as a direct benefit of directed testing method. In our example, a length of two would have sufficed.

3 Approach

Each tested program is a $VirgilProgram$ (see `http://compilers.cs.ucla.edu/virgil`); we compile each to *machineCode*, that is, AVR assembly code. A key input to each execution is an *eventSequence*—a list of tuples—where each tuple consists of an event-handler name (an *identifier*), a unique ID for each call to a handler (an *int*), a list of event argument values (as *ints*), and a *timeout*—the maximum time given for the execution of the event—measured in milliseconds. Each of our *constraints* is a Virgil arithmetic or logical expression, and a *solution* maps each relevant program variable (*identifier*) to an *int*, or is otherwise *None*. Our approach uses a data structure of type *state* that is a tuple of five values. If s is of type *state*, the first component ($s.wcet$) is the WCET found so far. The second ($s.coverage$) is the highest branch coverage found so far. The third ($s.eventSeq$) is the event sequence that led to WCET, and the fourth ($s.constraints$) the collected constraints during such pass. Finally the fifth component ($s.noChange$) is the number of consecutive rounds without improvements to either the WCET or branch coverage.

Tools. Our approach uses 7 tools, whose types are shown in Fig. 1: *Compiler* is an open-source Virgil compiler [16] that generates AVR machine-code. The tool *avrora* is an open-source simulator for AVR machine code [17]. The tool *random* takes no inputs and produces a random event sequence. The tool *timeoutCombos* takes an event sequence with undefined timeouts and duplicates the sequence for all possible combinations of timeout values for each event in the sequence. The tool *concolic* is a concolic execution engine for

$compiler : VirgilProgram \to machineCode$
$avrora : machineCode \times eventSequence \to wcet$
$random : () \to eventSequence$
$timeoutCombos : eventSequence \to (eventSequence\, list)$
$concolic\ :\ (VirgilProgram \times eventSequence) \to (wcet \times branchCoverage \times constraints)$
$solver : constraints \to solution$
$generator : solution \to eventSequence$

Figure 1 VICE tools.

```
Input:    $p$: VirgilProgram
Output:   wcet $\times$ branchCoverage $\times$ eventSequence
Local:    $a$: machineCode = compiler($p$), $s$: state = (0, 0.0, ( ), 0) , $roundId$: int = 0
          $seqs$: eventSequence = timeoutCombos(random())
Method:   while ($s.noChange < 2$) {
              foreach ($seq$ in $seqs$) {
                  let ($wcet$, $bc$, $c$) = concolic($p$, $seq$) in
                  $s$ = update($s$, $wcet$, $bc$, $c$, $roundId$)
              }
              $roundId$++
              $seqs$ = timeoutCombos(generator(solver($s.constraints$)))
          }
          return ($s.wcet$, $s.coverage$, $s.eventSeq$)
```

Figure 2 Event-based directed testing (EBDT) algorithm.

Virgil that takes a Virgil program and an event sequence. *Concolic* will run *avrora* to fire the events from the event sequence with the given set of timeout choices. The result of a run of *concolic* is a measurement of WCET, of the branch coverage achieved, plus a collection of constraints. *Solver* is a constraint solver used for the directed-testing approach. The tool *generator* takes a mapping of variable names to values and generates an event sequence.

Event-Based Directed Testing. Fig. 2 lists our algorithm. We compile each Virgil program to AVR assembly code. The algorithm starts from a randomly generated event sequence, and generates all combinations of timeout choices (sweeping a range) to produce a list of event sequences. For each sequence (which now has a particular choice for timeouts), it executes *concolic* on the Virgil program to get new values for the branch coverage, new constraints, as well as WCET on the assembly code by running *avrora*. The *solver* and *generator* will convert the constraints into a new event sequence to be used in the next round. After each run, we invoke an *update* function (omitted for brevity) which updates the state variable s to reflect the latest information of the WCET, branch coverage, and path constraints that led to WCET, found so far. We also update $s.noChange$ to reflect how many recent unique rounds with no change to either WCET or the branch overage have occurred. This is used as the terminating condition for the algorithm.

4 Experimental Results

We compare EBDT to random testing, genetic algorithms, and traditional directed testing.

4.1 Methodology

To have fair comparisons, we implemented random testing, a genetic algorithm, and traditional directed testing for Virgil. Our implementation of event-based directed testing—VICE[2]—is written in Java. We used an existing Virgil interpreter as the basis for our concolic execution engine, which tracks symbolic expressions alongside concrete values. For constraint solving, we use the open-source solver Choco [13]. We implemented the genetic algorithm on top of the Java genetic algorithm package (JGAP). In order to perform traditional directed testing with VICE, we ran our algorithm on each event in isolation and summed up the individual WCET estimates. For timeout values, we exhaustively sweep a range from the smallest allowed interval between context switches (measured 8 ms, empirically) up to the full execution time of the event handler when run without any interruptions.

4.2 Benchmarks

The following table shows some statistics about our eleven benchmarks that test device drivers for Berkeley Motes, including the Virgil and translated C lines of code counts. The C code is compiled into AVR assembly. The table also shows the number of event handlers.

Benchmark	LOC (Virgil)	LOC (C)	#handlers	Description
BinaryTree	114	167	1	a simple (unbalanced) binary tree
LinkedList	124	181	1	a simple doubly-linked list
BubbleSort	55	519	3	the common but slow bubblesort algorithm
Decoder	772	1015	3	decode instructions and other binary data
Oscilloscope	920	1338	11	a visualizer for sensor readings
Fannkuch	422	605	3	measures impact of compiler optimizations on runtime performance.
MsgKernel	773	1519	2	adaptation of the core message-passing mechanism from SOS operating system
TestRadio	1695	2833	6	tests the functionality of the Radio (wireless signal) driver.
TestUSART	1,226	1,737	5	tests the Universal Synchronous Asynchronous Receiver Transmitter driver.
TestSPI	859	1,109	3	tests the Serial Peripheral Interface driver.
TestADC	605	1,055	4	tests the Analog to Digital Converter driver.

4.3 Measurements

We performed our experiments on a 2.3 GHz Intel Core i7 iMac, with Sun Java2 SDK 1.5. All time measurements are in milliseconds. We used event sequences with 100 events for all experiments, except for traditional directed testing where each event sequence consists of a single event. In runs of the genetic algorithm, each generation has 500 event sequences. The genetic algorithm stops after two generations result in no improvement to the branch coverage or the WCET. We use the same number of event sequences for random testing and the genetic algorithm, for a fair comparison. Tab. 1 compares the results of EBDT and existing solutions, while Tab. 2 lists the timing of the runs. We also report the WCET found by Avrora's static deadline analyzer.

[2] VICE source with all benchmarks can be found at `https://github.com/Mah-D/VICE`.

Table 1 Experimental results: EBDT, plus static WCET.

	Random			**GA**			**DSE**			**VICE**			**Static**
Benchmark	#ES	WCET	BC	#ES	WCET	BC	#ES	WCET	BC	#ES	WCET	BC	WCET
BinaryTree	2000	14	26%	2000	15	41%	55	24	45%	1	25	45%	118
LinkedList	2000	20	29%	2000	20	48%	70	30	55%	1	30	55%	91
BubbleSort	1500	8	22%	1500	8	30%	13	13	43%	104	69	80%	153
Decoder	7000	21	20%	7000	24	34%	194	29	50%	499	62	73%	88
Oscilloscope	17500	39	7%	175000	47	19%	502	61	39%	1019	90	65%	555
Fannkuch	9000	15	18%	9000	19	29%	322	27	51%	717	81	76%	304
MsgKernel	7000	38	26%	7000	48	52%	172	50	63%	315	82	92%	218
TestRadio	11500	44	13%	11500	51	28%	249	63	40%	293	74	63%	323
TestUSART	8500	53	28%	8500	66	52%	279	73	72%	411	94	100%	176
TestSPI	9500	25	35%	9500	28	31%	44	37	43%	107	39	51%	401
TestADC	7000	14	33%	7000	22	65%	18	30	58%	336	41	97%	102

Table 2 Testing time: EBDT testing.

Benchmark	**Random**	**Genetic**	**DSE**	**VICE**	Benchmark	**Random**	**Genetic**	**DSE**	**VICE**
BinaryTree	8220	4187	40	2079	MsgKernel	7341	5710	48	1463
LinkedList	12641	8347	55	3753	TestRadio	6291	22080	40	928
BubbleSort	463	219	13	176	TestUSART	7812	4189	33	314
Decoder	10686	9118	64	2325	TestSPI	14432	11729	19	3271
Oscilloscope	41785	15959	66	7295	TestADC	6803	2460	11	893
Fannkuch	16804	12012	37	2104					

4.4 Assessment

WCET. Results are shown in Tab. 1. VICE found the longest WCET across random testing, genetic algorithm, and the traditional single event directed testing(DSE). Compare to random testing, genetic algorithm, and traditional directed testing VICE improves WCET by 203%, 176% , and 97%, respectively. Big differences are seen in BubbleSort and Fannkuch because of several nested conditional structures of these benchmarks. VICE's estimates were closest to the static (over-estimate) computation of WCETs in all benchmarks. None of the other testing approaches come close to match VICE's results consistently.

Branch coverage (BC). Tab. 1 also illustrates branch coverage results. For all benchmarks VICE gives the best result. For one benchmarks VICE gives 100% branch coverage, and in seven others VICE covered more than 70% of branches. BinaryTree shows the lowest coverage. This benchmark has numerous non-numeric conditionals which VICE cannot currently handle. None of the other approaches come close to match VICE's results consistently.

Number of event sequences (#ES). VICE achieves its results with significantly fewer event sequences than random testing and the genetic algorithm. For two benchmarks, the difference is about 4X, while for nine benchmarks, the difference is more than 10X. In contrast, VICE uses about 3X more event sequences on average than traditional directed testing with a single event per event sequence. These results show that VICE achieves good results with a fairly low number of event sequences.

Testing time. In almost all cases, VICE is significantly faster than random testing and the genetic algorithm, and slower than DSE.

The constraint solver. We found that CHOCO does a good job with number types while has poor support for other types and operations, e.g., array and bit operations, user types. Therefore, path constraints that aren't boolean or arithmetic were not collected by our concolic tool, hurting the branch coverage and subsequently WCET estimates.

Single event versus VICE. Traditional directed testing can be employed for finding WCET, by computing WCETs for each event in isolation and adding them as the total WCET of the program. Yet result are inaccurate and overly conservative. The conditions that result in WCET of individual event handlers may be impossible to have at the same time, since handlers can interrupt each other and interact indirectly via shared state. Our choice of event sequences of length 100 was chosen based on experience with the benchmarks. The more event handlers we have, the longer event sequences are needed for good testing. Finding suitable lengths of event sequences for each application is a subject of future work.

5 Related Work

In Sec. 4, we mentioned four event sequence generation techniques for WCET analysis: random technique [2], genetic algorithm [1], traditional directed testing [15, 7], and static analysis [3]. Here we highlight some of the notable techniques and tools in the area of WCET.

Static techniques examine the source code without running it and return an upper-bound for the true WCET. Different static techniques have been used to find WCET. aiT WCET [6], Bound-T [11], and SWEET [8] use *value analysis*, in which register and memory values are approximated, without running the program, at every program point. The results are necessarily conservative. For example, they perform poorly when loop structures are present. *Control-flow analysis* can be used (e.g., [9]) to determine possible execution paths, which aids WCET calculation. In control-flow analysis a superset of all execution paths is created with a control-flow graph. An input range and tasks will be passed as an input to the CFG, and the worst timing is computed. Performing control-flow analysis on source code is simpler than on machine code. However, compilation, code optimization, and linkage may change program control flow and make the analysis cumbersome. *Processor-behavior analysis*, where exact behaviors of a processor, as in memory accesses, caching, and pipelining, are mimicked, is employed by [10] for WCET calculation. In practice these tools' calculations related to the processor, memory hierarchies, buses, and peripherals are all approximates and thus their timing results are conservative.

Dynamic techniques , in which the program is executed, have been discussed in the previous section. These use a variety of ways to generate inputs, e.g., random and probabilistic techniques [2], genetic algorithms [1], or other heuristics [14]. The closest work to ours is pathcrawler [19], which is very similar to the single event traditional directed testing.

6 Conclusion

Testing of event-driven software is difficult because of the need for event sequences rather than single inputs. We have shown that a combination of random testing and directed testing can be used to automatically produce effective event sequences that are challenging to produce manually. We presented our approach—event-driven directed testing—as a major improvement over traditional directed testing, which also generally produces better results than random testing and genetic algorithms.

There are many opportunities for future work. More general classes of path constraints need to be supported. Constraint solving may also be useful for both generating schedules (we currently use randomly generated sequences), as well as effectively finding suitable interrupt timeout values. These improvements will make coverage and WCET estimates more accurate, while reducing analysis times.

References

1 Austrian Computer Society (OCG). *Heuristic Worst-Case Execution Time Analysis.* 10th European Workshop on Dependable Computing, 1999.

2 Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS'02, pages 279–, Washington, DC, USA, 2002. IEEE Computer Society.

3 Dennis Brylow and Jens Palsberg. Deadline analysis of interrupt-driven software. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 198–207, New York, NY, USA, 2003. ACM.

4 Mahdi Eslamimehr and Jens Palsberg. Testing versus static analysis of maximum stack size. In *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, COMPSAC'13, pages 619–626, Washington, DC, USA, 2013. IEEE Computer Society.

5 Mahdi Eslamimehr and Jens Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 353–365, New York, NY, USA, 2014. ACM.

6 Christian Ferdinand. aiT: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

7 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'05, pages 213–223, New York, NY, USA, 2005. ACM.

8 Jan Gustafsson and Andreas Ermedahl. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 57–66. IEEE, 2006.

9 Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 287–297. IEEE, 2005.

10 Reinhold Heckmann. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

11 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP*, 457:307–312, 2000.

12 M. Kirsch. Technical support to the national highway traffic safety administration (NHTSA) on the reported Toyota motor corporation (TMC) unintended acceleration (UA) investigation. Technical report, NASA, 2011.

13 François Laburthe et al. Choco: implementing a CP kernel. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, volume 55, pages 71–85, 2000.

14 Peter Puschner and Roman Nossal. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 134–143. IEEE, 1998.

15 Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE'07, pages 571–572, New York, NY, USA, 2007. ACM.

16 Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA'06, pages 191–208, New York, NY, USA, 2006. ACM.

17 Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN'05, Piscataway, NJ, USA, 2005. IEEE Press.

18 Reinhard Wilhelm, Jakob Engblom, and Andreas Ermedahl. The worst-case execution-time problem; overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

19 Nicky Williams and Muriel Roger. Test generation strategies to measure worst-case execution time. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 88–96. IEEE, 2009.

GenE: A Benchmark Generator for WCET Analysis

Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

The fact that many benchmarks for evaluating worst-case execution time (WCET) analysis tools are based on real-world applications greatly increases the value of their results. However, at the same time, the complexity of these programs makes it difficult, sometimes even impossible, to obtain all corresponding flow facts (i.e., loop bounds, infeasible paths, and input values triggering the WCET), which are essential for a comprehensive evaluation. In this paper, we address this problem by presenting GenE, a benchmark generator that in addition to source code also provides the flow facts of the benchmarks created. To generate a new benchmark, the tool combines code patterns that are commonly found in real-time applications and are challenging for WCET analyzers. By keeping track of how patterns are put together, GenE is able to determine the flow facts of the resulting benchmark based on the known flow facts of the patterns used. Using this information, it is straightforward to synthesize the accurate WCET, which can then serve as a baseline for the evaluation of WCET analyzers.

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

Keywords and phrases WCET, benchmark generation, flow facts, WCET Tool Challenge

Digital Object Identifier 10.4230/OASIcs.WCET.2015.33

1 Introduction

Benchmarks such as Mälardalen [11], DEBIE-1 [13], or PapaBench [19] play an important role in the evaluation of WCET analyzers as they allow to assess the individual strengths and weaknesses of different tools, for example, in the context of the WCET Tool Challenge. If the focus of such an evaluation lies on the comparison of two or more WCET analyzers, having the source code and/or binaries of a benchmark available is usually sufficient to achieve meaningful results. However, for a comprehensive investigation, additional questions about the properties of the program under test need to be answered in order to be able to objectively assess the quality of a particular WCET analyzer:

1. What are the loop bounds?
2. What are the feasible paths?
3. What are the recursion depths?
4. What are the concrete input values triggering the WCET?

Throughout this paper, we refer to such information as the flow facts of a benchmark. Their significance stems from the fact that flow facts can serve as an absolute baseline for the validation of WCET-analyzer results, thereby enabling evaluations that go beyond the relative comparison of multiple tools. Note that depending on the analysis approach used, some flow facts are more important than others: while loop bounds, feasible paths, and recursion depths are especially relevant in the context of a static timing analysis, having available concrete input values triggering the WCET, for example, is of major importance for measurement-based timing analysis. With knowledge about the concrete worst-case input values, which also take hardware features (e.g., caching) into account, the actual WCET of a benchmark can be determined by concretely executing it on a cycle-accurate simulator or on the specific target platform while measuring its execution time.

Unfortunately, extracting all possible flow facts from existing benchmarks is inherently difficult, in some cases even impossible: From a theoretical point of view, non-trivial properties of program code are not automatically decidable [22]. This applies also to existing benchmark suites [11, 13, 19] written in high-level programming languages.

Knowledge about all existing flow facts requires knowledge about all existing program paths, which makes it necessary to explicitly enumerate these paths. Such an explicit path enumeration is infeasible for most real-world programs due to the myriad of possible paths [17]. Furthermore, manually determining all possible flow facts of programs is labor-intensive and error-prone [3].

In this paper, we address the problem of determining flow facts for WCET benchmarks by presenting GenE[1], a tool that provides both WCET benchmarks as well as their flow facts. GenE is able to achieve this, because instead of analyzing existing programs, the tool generates new benchmarks by relying on small building blocks (for which the flow facts are known) and combining them in a way that allows the tool to automatically determine the flow facts of the resulting complex benchmark. In order to create realistic benchmarks, the building blocks used by GenE are typical design and implementation patterns that have been extracted from existing real-world applications and WCET benchmark suites.

In particular, the contributions of this paper are:

- GenE, a tool to automatically generate benchmarks, enabling users to conduct comprehensive evaluations of WCET analyzers.
- An algorithm to create complex WCET benchmarks from program patterns that allows to keep track of flow facts.
- A discussion of how the flow facts provided by GenE can be used to determine the WCET of a generated benchmark for both static and measurement-based timing analysis.

The remainder of this paper is structured as follows: Section 2 discusses a number of general principles by which the evaluation of a WCET analyzer should be guided. Section 3 presents details of GenE and explains how our tool enables users to meet these principles. Section 4 provides an overview of related work, and Section 5 concludes and gives an outlook on future work.

2 A Vision for the Comprehensive Evaluation of WCET Analyzers

Conducting a comprehensive evaluation of WCET analyzers is a challenging task. In the following, we identify a number of requirements that are crucial in this context and discuss whether, and if so to which extent, they are met by existing approaches.

Large Number of Benchmarks. In order to get stable and comparable results, WCET analyzers should be evaluated using a large set of different benchmarks. This minimizes the risk that a few poorly-selected benchmarks lead to false conclusions.

In practice, the evaluation of WCET analyzers is usually performed based on an existing benchmark suite [11, 12, 19, 20] or a set of manually-selected programs. As a consequence, the number of benchmarks used typically ranges between one and around twenty. While this may be sufficient for some use cases, in general it is desirable to have a greater number of

[1] The name GenE originates from the term genie and the idea of **Gen**erating **E**valuation sets.

benchmarks. Considering the effort it takes to select/implement a benchmark, we argue that this goal can only be achieved by generating benchmarks automatically.

Realistic Benchmarks. WCET analyzers should be evaluated under real-world conditions. In consequence, in the optimal case, the benchmarks used are actual applications or at least closely resemble them.

Many benchmark suites available have been composed with a focus on real-world programs and consequently meet this requirement. Of course, this is also implicitly true for evaluations that are performed based on actual applications.

Wide Spectrum of Benchmarks. An evaluation should take into account that different applications have different characteristics, in particular, with regard to complexity and program structure. For example, while state-machine-like applications are usually built around a central control loop, signal-processing applications in contrast often consist of a static input array and nested loops. Introducing variety into the set of benchmarks offers the possibility to properly reflect such differences. Furthermore, it allows to better compare the individual strengths and weaknesses of the WCET analyzers evaluated.

The state-of-the-art Mälardalen WCET benchmarks [11] and the TCAS benchmark [9], which was used at the WCET Tool Challenge 2014, cover both of the categories mentioned above: state-machine-like applications (i.e., `statemate`, `tcas`) as well as signal-/image-processing programs (i.e., `fir`, `edn`, `jfdctint`).

Preservation of Benchmark Properties. The selection procedure must take into account that compiler optimizations may have significant impact on programs. As a result, the timing analysis either needs to be performed on an already optimized representation of the benchmark or needs to ensure that the benchmark properties are preserved by the compiler across code transformations.

Having investigated several publicly-available benchmarks, we found that without further modifications a significant number of Mälardalen WCET benchmarks is not resistant to compiler optimizations, by default, without setting additional preprocessor definitions: For example, Clang optimizes five of these benchmarks (i.e., `bs`, `expint`, `fibcall`, `janne_complex`, `ns`) to a single return statement. This is due to values stored in variables that have no effect on the output enabling compilers to decisively change the structure of the program.

Availability of Benchmark Flow Facts. In order to be able to independently assess the quality of a WCET analyzer, the flow facts of a benchmark should be known prior to the evaluation. This way, a WCET analyzer can be rated based on how close its outputs are to the actual results of a benchmark.

In general, there is a tradeoff between the demand to have realistic benchmarks and the requirement to obtain flow facts, as precise numbers are usually not available for real-world applications for reasons of complexity (see Section 1). As a consequence of the fact that, as discussed above, many existing benchmarks focus on realistic use cases, evaluations are therefore mostly based on a comparison of results, or rely on an absolute baseline that is either determined manually or an over-approximation [3, 17]. That is, in order to evaluate the impact of a novel feature, the WCET analysis is performed twice: once with the feature enabled and once with the feature disabled. Although such an approach allows to assess the relative improvement achieved by the feature, it is not suitable to determine the benefit of the feature on a global scale.

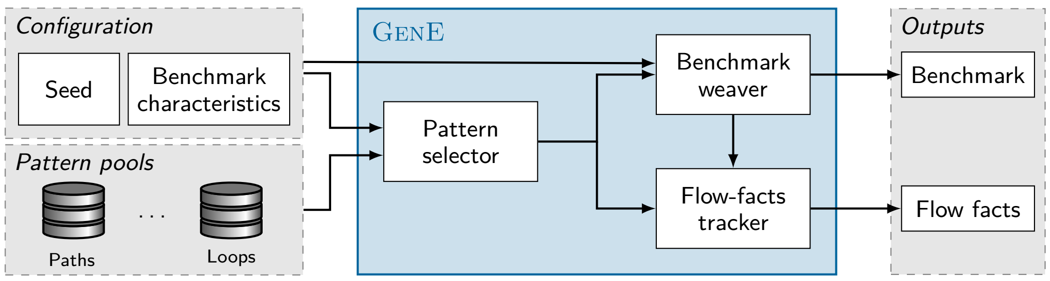


Figure 1 GenE automatically generates benchmarks by combining existing program patterns.

3 GenE

In this section, we present GenE, an approach and tool to address the problem of providing benchmarks for the evaluation of WCET analyzers. GenE generates benchmarks automatically and therefore allows a **large number of benchmarks** to be used, as they no longer have to be selected or implemented manually. In order to provide **realistic benchmarks**, the tool combines common building blocks (e.g., conditional statements and nested loops) we extracted from real-world applications. By varying the selection and composition of patterns, GenE supports the creation of a **wide spectrum of benchmarks**. To assemble a new program, GenE relies on a deterministic procedure designed to ensure the **preservation of benchmark properties**. In addition, this procedure also allows the tool to determine the flow facts of a benchmark based on knowledge about its structure, enabling GenE to guarantee the **availability of benchmark flow facts** for all programs created.

Figure 1 shows an overview of the GenE workflow: When a user requests the tool to create a benchmark with certain characteristics (e.g., those of a signal-processing application), the tool first selects suitable programs patterns from a set of pattern pools (see Section 3.1). In the next step, GenE composes the actual benchmark by weaving the selected patterns into a program (see Section 3.2). Based on the information which patterns have been selected as well as knowledge about how they have been put together, the tool is then able to track the flow facts for the generated benchmark (see Section 3.3).

3.1 Patterns and Pattern Selection

When generating a new benchmark, the first step performed by GenE is to automatically select different building blocks, which in a later step are then combined to a complex program. In order to enable the tool to create realistic benchmarks, we conducted a study of existing programs used for the evaluation of WCET analyzers to identify recurring patterns. The results of this analysis are a combination of design and implementation patterns described in literature [7] as well as patterns directly extracted from the code of state-of-the-art benchmark suites such as Mälardalen [11]. Listings 1 and 2 present two simple examples of patterns used by GenE: a path pattern including a conditional branch and a parameterizable nested-loop pattern. As shown in the listings, both patterns comprise a number of *insertion points*, which allows GenE to combine them with other patterns during the benchmark-weaving step (see Section 3.2).

GenE manages the building blocks of benchmarks in a set of pattern pools, thereby grouping together patterns with similar structure (e.g., array accesses in loops). Users are able to implement new patterns and add them to existing pools; in addition, the tool also offers the possibility to introduce new pattern pools. Pattern pools play an important role during

Listing 1 Example of a path pattern.

```c
if( condition ){
  // insertion_point_1
} else {
  // insertion_point_2
}
// insertion_point_3
```

Listing 2 Example of a loop pattern.

```c
for(i = n-1; i >= 1; i--){
  for(j = 0; j < i; j++){
    // insertion_point_1
  }
}
// insertion_point_2
```

the pattern-selection step: By assigning different weights to different pools, GenE ensures that the benchmark generated matches the properties specified by the user. For example, if a user requests a program matching the characteristics of a digital-signal-processing application, the tool favors the selection of loop patterns operating on arrays, as such patterns are typical for this category of use cases.

3.2 Benchmark Weaving

After the patterns have been selected, they are combined during the benchmark-weaving phase to create new complex benchmarks. To address the problem of tracking flow facts (see Section 3.3), GenE uses a formal grammar G that is considered by the weaving algorithm. The grammar G contains the start symbol (S) and the empty string (ε). A point in the control-flow graph of the generated benchmark where further expansions through statements (e.g., assignments, loops, branches) are possible is called an insertion point (*inp*) (see Listing 1 and 2). The following list is an excerpt of the production rules of G:

1. $\mathcal{S} \mapsto \mathit{FunctionBegin} \cdot \mathit{inp} \cdot \mathit{FunctionEnd}$
2. $\mathit{inp} \mapsto \varepsilon$
3. $\mathit{inp} \mapsto \mathit{Statement} \cdot \mathit{inp}$
4. $\mathit{Statement} \mapsto \mathit{Assignment}$
5. $\mathit{Statement} \mapsto \mathit{IfBegin} \cdot \mathit{inp} \cdot \mathit{ElseBegin} \cdot \mathit{inp} \cdot \mathit{EndIf}$
6. $\mathit{Statement} \mapsto \mathit{LoopHead} \cdot \mathit{inp} \cdot \mathit{LoopEnd}$

$\vdots$ *(Additional production rules)*

The start symbol is mapped to exactly one insertion point (1), which is surrounded by the begin and end of the main function. Each insertion point can produce an empty element (2) or a statement followed by a further insertion point. Production rule 3 ensures the sequential creation of patterns. The nonterminal *Statement* symbols can produce assignments, branches, or loops (4–6).

The grammar describes what benchmarks can be produced whereas the weaving algorithm presented in Listing 3 exemplarily illustrates how GenE uses this grammar to create a benchmark. The weaving algorithm recursively inserts new patterns into insertion points. The core concept of the algorithm is to bound the timing cost for inserted patterns from top to bottom. There can be program paths through this pattern that are less expensive than the worst-case path[2], but the worst-case path must exactly lead to the predefined cost. All operations in Listing 3 based on a seed value are prefixed with `select_`, concrete insertions of code into the current selected insertion point are prefixed with `emit_`, and the `costof()` function returns the cost of sequential statements. While descending the tree, the costs are subdivided and used for further productions.

[2] It is assumed that the worst-case path is defined only through its concrete input values and no other effects like concurrency or non-deterministic input/output operations.

Listing 3 The recursive algorithm of GenE inserts patterns top-down into the benchmark.

```
 1 function GenE(vars, cost)
 2  if(cost == 0) return vars
 3
 4  switch(select_production(cost))
 5    case Statement · inp:
 6       statement_cost ← select_cost ∈ [0, cost]
 7       new_vars ← GenE(vars, statement_cost)
 8       new_vars ← GenE(new_vars, cost - statement_cost)
 9    case Assignment:
10       (new_vars, operation) ← select_assignment(vars)
11       emit_Assignment(operation)
12    case IfBegin · inp · ElseBegin · inp · EndIf:
13       condition ← select_condition(vars)
14       emit_IfBegin(condition)
15       if_cost ← cost - costof(condition) // if-case is worst case here
16       else_cost ← if_cost - (select_cost ∈ [0, if_cost])
17       if_vars ← GenE(vars, if_cost)
18       emit_ElseBegin()
19       else_vars ← GenE(vars, else_cost)
20       emit_EndIf()
21       new_vars ← merge_variables(if_vars, else_vars)
22    case LoopHead · inp · LoopEnd:
23      ...
24  return new_vars // return updated variables including value ranges
```

The algorithm receives a list of available variables and the timing cost that must be produced by the emitted code for the respective call of the function (Line 1). If no cost is available for productions, the algorithm returns the current variables including their values (Line 2), which is the termination criterion for the algorithm. Otherwise, a production is selected based on the distributable cost (Line 4).

Cost Distribution. The cost is subdivided into parts based on the seed. The parts are used for generating a statement followed by an insertion point (Line 5–8). The concrete cost modeling, which is necessary for determining worst-case paths including their input values, is discussed in Paragraph 3.3.1.

Recursive Application. After determining the cost for the statement, GenE is recursively applied (Line 7) with the respective cost. The rest of the distributable cost is used for a second recursive call of the GenE function enabling sequences of pattern insertions.

Value Tracking. For computed values that are used, for example, for branch conditions, the values of all variables and their availability must be tracked. For example, the *Assignment* production can introduce a new variable that is computed from two further variables (Line 10). Consequently, the updated variables are returned at the current insertion point during the code-generation process.

3.3 Flow-Facts Tracking

Tracking the value ranges of variables is an essential mechanism for the flow-facts tracking that takes place during the benchmark-weaving phase. These flow-facts include, for example, feasible paths, loop bounds, or values triggering the most expensive path. The concrete input value leading to the WCET is determined prior to the first call of the GenE function based on the seed value. Consequently, all costs of feasible branches in the control flow must be modeled according to the input variable taking computations on the input into consideration.

Reconsidering Listing 3, when selecting the branch condition during the application of the *If-Else* production (Line 13), the current values of the used variables are taken into account and the costs of the branches (Line 15–16) are set according to the worst-case input. The cost of each branch is again exactly bounded through the recursive calls of GenE (Line 17, 19). The possible values of the variables are merged after emitting the *If-Else* pattern and the values for the worst-case path are stored (Line 21).

The patterns hold their own (parameterizable) flow facts, for example, as formulas that must be determined when the pattern is manually constructed. These formulas are considered when inserting the pattern and combined with the existing flow facts. For example, reconsidering the parameterizable loop pattern shown in Listing 2, the loop bound of the outer loop is `n-1` whereas the parametric formula for the inner body is `n*(n-1)/2`. These formulas are used when the pattern is woven into the code and the concrete flow facts are determined with the concrete value of the variable `n`.

3.3.1 Cost Modeling

WCET analysis is typically split into two parts: a high-level analysis of flow facts and a low-level analysis of the processor (i.e., instruction execution times, cache modeling). GenE focuses on generating benchmarks with challenging, high-level flow facts and not on difficult access patterns for cache analysis for a specific architecture. However, to distribute cost for patterns, knowledge on timing costs of the target platform is necessary. GenE addresses the cost-modeling problem through *relatively* modeling timing costs [23] and *overweighting* the branches of determined worst-case paths with these relative costs of instructions.

Reconsidering the *If-Else* path pattern shown in Listing 1, the branch of the *If* case is overweighted by a large factor. This factor for overweighting worst-case branches must ensure that even if the *Else* branch only creates cache misses, the *If* branch is still more time consuming. Furthermore, since GenE is implemented on a low-level intermediate representation of program code (see Section 3.4), this representation can be attributed with concrete cost information [6, 10] to further refine the cost model through target-specific knowledge allowing smaller factors for overweighting the worst-case path.

However, for comparing the precision of WCET-analysis tools, an absolute WCET value must be known for the generated path conditions. A discussion of how the flow facts provided by GenE can be used to determine the WCET is described in the following paragraph.

3.3.2 Determination of the WCET

Detailed knowledge about the input values triggering the worst-case path is stored through the value and flow-facts tracking. Consequently, the generated benchmark can be concretely executed with the determined worst-case input leading to the actual WCET when measuring the time of this execution. The concrete execution and precise measurement can be achieved through two approaches. First, modern processors are equipped with highly accurate internal time-stamp counters that allow precise time measurements without the need of external measurement hardware. Second, if a cycle-accurate simulator for the targeted processor is available, the actual WCET can be derived through a simulation, without requiring the target hardware. Once the actual WCET of the generated program is determined, it serves in combination with the known flow facts as a baseline for the evaluation of both static and measurement-based timing analyzers. Precisely modeling the behavior of the processor is not mandatory in such an input-oriented approach as long as the determined input values lead to the WCET when executing the program with them, which is achieved through the relative overweighting of branches. Consequently, complex hardware features (e.g., caching) are implicitly respected through the concrete execution.

3.4 Implementation

GenE is implemented with the framework of the Low Level Virtual Machine (LLVM) [18] using its intermediate representation (LLVM-IR) for the following reasons:

- The generated benchmarks are independent from specific processors to generate machine code for various targets. The cost modeling is performed on basis of the LLVM and is further improvable through target-specific knowledge [6, 10].
- Although the LLVM-IR is independent of specific target machines, it can be precisely mapped to machine code through control-flow relation graphs [14]. Using such graphs, the knowledge of flow facts can be transformed from the LLVM-IR level to machine code.
- If the language of the generated benchmarks is implemented on a higher abstraction level, it is more difficult to track the control-flow relations during optimizing source-to-source transformations. However, GenE focuses on generating benchmarks that look like optimized code, since such code facilitates timeliness and unoptimized code is unrealistic for real-world scenarios. To handle this problem, GenE uses optimized patterns that are further combined. During lowering the LLVM-IR to machine code, the generated benchmark must not be optimized further and must not change its control flow.

GenE implements several challenging program patterns taken from [7, 11] and is extensible through the integration of additional patterns. Further benchmark suites, where patterns with different characteristics are extractable for weaving new benchmarks, are MiBench [12], BEEBS [20], or PapaBench [19].

To evaluate the effectiveness of our benchmark-generation algorithm, we run GenE with different initial seeds and maximum distributable cost leading to different execution times. With this configuration, GenE is able to generate 10,000 benchmarks within four minutes on an Intel Core-i7 (8 GB RAM). Although using a single integer input value with 32 bits, 2^{32} control flows through the generated benchmark can be produced. This input space is considered to be sufficiently large for WCET benchmarking.

4 Related Work

GenE is the first approach to automatically generate benchmark programs specifically targeting the evaluation of WCET-analysis tools. However, the development of this benchmark generator is strongly inspired by the Csmith [24] benchmark generator. Csmith targets the domain of testing compilers for their correctness, for example, through generating complex branch expressions. The concept of inserting patterns into existing code recursively top-down and thereby creating new insertion points is similar to the recursive expansion process of Dujmović [8] for generating performance benchmarks. Additionally to this approach, GenE tracks low-level flow facts and worst-case input values targeting the WCET domain. In the domain of high-performance computing, the MicroProbe framework [2] is an automated approach to micro-benchmark generation. The generated benchmarks are, for example, utilized for energy-related characterization of a specific target architecture. We consider benchmarks generated by GenE as suitable source for comparing the effectiveness of existing WCET-analysis tools [1, 5, 16, 21].

To compare best the flow facts found by WCET analyzers, GenE must support to output flow facts in a format that analyzers under evaluation can process, which is considered future work. An example for such a flow-fact language is FFX [4]. Further discussions on suitable annotation languages to provide flow facts are found in [15].

5 Conclusion and Outlook

Comparing WCET-analysis tools through benchmarking is a challenging task since many aspects must be considered, such as providing a wide spectrum and large number of benchmarks for which the flow facts are available. We therefore propose the GenE benchmark generator that is able to automatically generate benchmarks with diverse characteristics including their flow facts. With knowledge about these flow facts and input values, a precise WCET value can be determined of the generated benchmark, which serves as a common baseline for timing-analysis tools. Therefore, we consider to use benchmarks built by GenE together with generated flow facts, the input assignment, and the WCET as a suitable evaluation scenario for upcoming WCET Tool Challenges, in addition to existing benchmarks.

Especially for the upcoming challenges in multi-core WCET analysis of complex concurrency patterns, the GenE approach is useful to create a known baseline. This is due to the fact that these benchmarks are more complex compared to benchmarks on single-core processors. Consequently, manually determining the WCET of multi-core benchmarks is inherently more labor-intensive than of single-core benchmarks. An incremental process is imaginable where new challenging concurrency patterns are progressively integrated into GenE and WCET tools catch up with the analysis of these patterns. To improve the existing prototype of GenE and to make it applicable for use, GenE will be made available after incorporating feedback from the WCET community.

Acknowledgements. This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-1 (Aspect-Oriented Real-Time Architecture), Research Grant no. SCHR 603/13-1 (Power-Aware Critical Sections), and the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89, Project C1).

References

1 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

2 R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose. Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 199–211, 2012.

3 B. Blackham, M. Liffiton, and G. Heiser. Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets. In *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, pages 169–178, 2014.

4 A. Bonenfant, H. Cassé, M. De Michiel, J. Knoop, L. Kovács, and J. Zwirchmayr. FFX: A portable WCET annotation language. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 91–100, 2012.

5 A. Bonenfant, M. de Michiel, and P. Sainrat. oRange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, 2008.

6 C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 17th International Symposium on Low Power Electronics and Design*, pages 333–338, 2011.

7 D.-H. Chu and J. Jaffar. Symbolic simulation on complicated loops for WCET path analysis. In *Proceedings of the 9th International Conference on Embedded Software*, pages 319–328, 2011.

8 J. Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 263–274, 2010.

9 A. Gotlieb. TCAS software verification using constraint programming. *The Knowledge Engineering Review*, 27(03):343–360, 2012.

10 N. Grech, K. Georgiou, J. Pallister, S. Kerrison, and K. Eder. Static energy consumption analysis of LLVM IR programs. *Computing Research Repository, arXiv*, pages 1–12, 2014.

11 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 137–147, 2010.

12 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 *Proceedings of the International Workshop on Workload Characterization*, pages 3–14, 2001.

13 N. Holsti, T. Langbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *Proceedings of the Data Systems in Aerospace Conference*, pages 1–6, 2000.

14 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proceedings of the 14th Conference on Languages, Compilers and Tools for Embedded Systems*, pages 163–172, 2013.

15 R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET analysis: The annotation language challenge. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*, pages 1–17, 2007.

16 J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop bounds for WCET analysis. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 435–444, 2012.

17 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 161–170, 2013.

18 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

19 F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, pages 1–6, 2006.

20 J. Pallister, S. J. Hollis, and J. Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *Computing Research Repository, arXiv*, pages 1–12, 2013.

21 P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proceedings of the 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 1–8, 2013.

22 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

23 P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 1–10, 2015.

24 X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation*

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

1 **Fachgebiet Rechnersysteme, Technische Universität Darmstadt, Germany**
`{dreyer,hochberger}@rs.tu-darmstadt.de`

2 **AbsInt Angewandte Informatik GmbH, Germany**
`wegener@absint.com`

3 **Accemic GmbH & Co. KG, Germany**
`aweiss@accemic.com`

Abstract

Precise estimation of the Worst-Case Execution Time (WCET) of embedded software is a necessary precondition in safety critical systems. Static methods for WCET analysis rely on precise models of the target processor's micro-architecture. Measurement-based methods, in contrast, rely on exhaustive measurements performed on the real hardware. The rise of the multicore processors often renders static WCET analysis infeasible, either due to the computational complexity or due the lack of necessary documentation. Current approaches for (hybrid) measurement-based WCET estimation process the trace data offline and thus need to store large amounts of data. In this contribution, we present a novel approach that performs continuous online aggregation of timing measurements. This enables long observation periods and increases the possibility to catch rare circumstances. Moreover, we incorporate the execution contexts of basic blocks. We can therefore account for typical cache behaviour, without being overly pessimistic.

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

Keywords and phrases hybrid worst-case execution time (WCET) estimation for multicore processors, real-time systems

Digital Object Identifier 10.4230/OASIcs.WCET.2015.45

1 Introduction

Today, embedded systems are a central part of almost all technical systems. In safety critical systems, proper function does not only rely on correct internal sequence of operations, but also on the timing of the operations. Particularly in real-time systems, upper bounds for the computation of system responses must be given.

Traditionally, the term "Worst-Case Execution Time (WCET)" is used to describe the timing properties of the code under scrutiny. In general, however, the WCET cannot be computed precisely but must be estimated. Such an estimate is only safe if the estimation process is guaranteed to never underestimates the execution time.

One method to compute safe upper bounds of the execution time is the abstract interpretation of the code in question. Each and every instruction is simulated on the basis of a

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

precise processor model. This type of estimation gives very good results, if the features of the processor are known and predictable.

Unfortunately, modern processors often contain features with unpredictable behaviour, and sometimes the documentation of the architecture is not precise enough. This can yield overly pessimistic estimations. One source of such uncertainties are bus arbitration policies. These are particularly difficult for multicore processors, where several cores share the same memory. Prominent examples of such processors include the P4080 from Freescale (eight cores, 1.3 GHz each) and systems based on the ARM Cortex A9. Recent research (e.g. [11]) investigates approaches to mitigate these problems, but no general solution exists so far to perform static WCET analysis for multicore processors.

Other methods to estimate the execution time use measurements. Their safety depends on the premise that the worst-case is actually observed. Current measurement-based approaches either rely on instrumentation or on the offline analysis of execution traces. Code instrumentation is usually not considered [12], as it causes the probe effect. Offline analysis of the trace data requires a sufficiently long recording of the traces to hit all relevant cases and system situations. Hence such systems are limited. Even if the recording depth is large, the offline processing time is high.

In this contribution, we show that continuous aggregation of execution time data per basic block can be achieved, relieving us of the tedious task of offline analysis. The statistics created by our approach can even account for cache effects by telling apart initial and subsequent executions of loop bodies. Together with an ILP-based path analysis, we achieve precise execution time estimates for complex multicore processors where existing approaches failed so far.

2 Related Work

For the sake of brevity, we focus on measurement-based and hybrid methods. We refer to [16] for a more complete overview of existing methods and tools.

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

To overcome this, more structured approaches have been proposed, e.g. by Betts et al. in [4], which combine the measured execution times of small code snippets to form an overall execution time estimate. Their use of software instrumentation leads to the *probe effect*, i.e. the timing behaviour of the program under observation changes due to the used observation technique. Moreover, their method does not account for typical cache behaviour and may be overly conservative.

In a more recent publication [5], they use the non-intrusive tracing mechanisms of state-of-the-art debugging hardware. The main obstacle of their method is the limited size of trace buffers and/or the huge amount of trace data. According to their estimates, around half a terabyte of data would be generated in an hour of testing.

Stattelmann et al. [13] propose the use of context information in order to account for cache effects. Their work shows that the inclusion of context information leads to more precise execution time results. However, their approach is limited to processors with very sophisticated tracing mechanisms. Moreover, their approach faces the same challenge as the ones above due to its offline analysis of trace data.

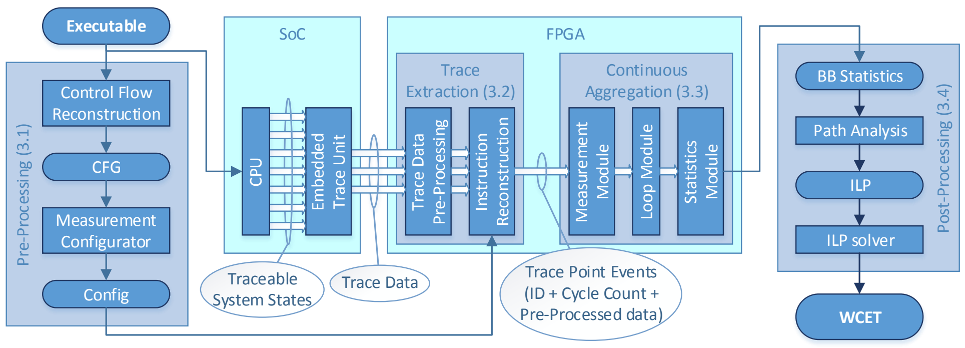


Figure 1 The workflow of our proposed method.

Other works have been done in order to overcome the obstacle of exhaustive measurements, e.g. [10, 17, 6]. They either generate input data [10, 17] to stimulate high path coverage or try to reduce the timing variability of a program [6].

There are many more publications on measurement-based execution time analysis which we do not list here. Overall, they either address the problem of generating suitable input data (which is out of scope of this paper) or they have one or several major disadvantages:

- The measurements are not fine-grained enough and hard to interpret.
- The use of software instrumentation leads to the probe effect.
- Huge amounts of trace data is generated for offline analysis.
- Disregarding the execution context leads to overly conservative results because cache effects cannot be exploited.
- The use of sophisticated tracing mechanisms limits the applicability to few selected processors.

Our method, in contrast, circumvents these drawbacks:

- We measure the timing of basic blocks. This allows to see where time is spent.
- We use non-intrusive hardware tracing mechanisms of state-of-the-art processors to produce timestamps. The probe effect is avoided.
- We process the trace events online. There is no need to store huge amounts of trace data.
- We process the trace events continuously. The aggregation can literally run for weeks. The possibilities to catch rare circumstances are increased.
- We incorporate the execution context of a basic block and account for typical cache behaviour. The results are thus much more precise.
- Due to the use of an FPGA, we can adapt the hardware part of our method to lots of different processors as long as they have some rudimentary tracing support.

3 Proposed Method

This section presents a novel approach for hybrid measurement-based timing analysis. The basic idea is to use an FPGA to perform online aggregation of trace data. Our method works on the object code level and is split into three main parts: a pre-processing phase, the continuous online aggregation phase and a post-processing phase. The workflow of our method is shown in figure 1. Parts are re-used from the `aiT` tool chain [1], in particular the control flow reconstruction and the ILP-based path analysis. Other parts, like the FPGA-based continuous online aggregation of measurements, are – to our knowledge – novel.

First, in the pre-processing phase, we reconstruct the interprocedural control flow graph (CFG) of the program under analysis. Then, we extract crucial information like loop nesting levels, call relations and basic block boundaries from the CFG. This information is used to compile a configuration for the trace extraction module which is then loaded into memory that is connected to the FPGA.

During the program's execution, the trace extraction module emits events according to its configuration. A possible event is, for example, the entering of a basic block. All events have an associated timestamp. The measurement module interprets these events and calculates the execution time of each basic block. It uses the precalculated information to keep track of the execution contexts of the basic blocks. Various statistics (minimum runtime, maximum runtime, sum of the individual runtimes, count of executions) are updated each time after a basic block timing measurement has been completed.

After the program has finished (or the test engineer has collected enough data), the post-processing phase is started by downloading the basic block statistics from the statistics module. Subsequently, the CFG together with the basic block timing statistics are used to construct an integer linear program (ILP). Solving this ILP gives then a path with the longest execution time (and consequently, an estimate of the worst-case execution time).

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

3.1 Control Flow Reconstruction and Pre-Processing

The starting point of our analysis is a fully linked binary executable. A parser reads the output of a compiler and disassembles the individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. With this knowledge, the binary reader forms the basic blocks of the CFG. A basic block (BB) is a sequence of instructions, where each instruction except the first and the last has exactly one predecessor and one successor.

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

The final result is an interprocedural control flow graph, as shown in figure 2. It consists of the basic blocks, some meta blocks to emphasise call/return relations and the edges that describe the control flow. [15] contains a detailed description of the control flow reconstruction process.

In a second step, we extract the information from the CFG that is needed for the configuration of the trace extraction module. To do so, we construct the set of trace points $\mathbb{T}$.

Besides the information needed to keep track of the execution contexts of the basic blocks, we also need to create unique identifiers for each basic block that are used to address the statistics in memory. Using the address of a block is not suitable, because we need to compact the address range of the statistics storage because memory is scarce. A collision-free hash function T-ID : $\mathbb{T} \rightarrow \{1, \ldots, n\}$ with $n \in \mathbb{N}$ is used for this purpose. It should be fast computable in hardware.

Each trace point (*first*, *tag*, *level*, *distinctor*, *call*, *entry*, *exit*, *return*) $\in$ $\mathbb{T}$ represents a basic block in the CFG and consists of the following data fields:

- *first*, the address of the BB's first instruction;
- *tag*, a custom field that can be used by function T-ID to calculate the ID of that trace point;

- *level*, the loop nesting level of the loop to which the BB belongs ($level = 0$ iff the BB does not belong to a loop);
- *distinctor*, a value used to distinguish two loops with the same nesting level that occur directly after another, without any code in between that does not belong to one of the two loops;
- *call*, a flag that indicates whether the BB is the predecessor of a call block;
- *entry*, a flag that indicates whether the BB is the successor of a routine's entry block;
- *exit*, a flag that indicates whether the BB is the predecessor of a routine's exit block;
- *return*, a flag that indicates whether the BB is the successor of a return block.

Moreover, due to the restricted bandwidth of the event stream, the fields of a trace point need to fulfill the following size limitations:

$$\begin{aligned}\forall p = (\mathit{first}, \mathit{tag}, \mathit{level}, \mathit{distinctor}, \dots) \in \mathbb{T} : & \; 0 \leq \mathit{first} \leq (2^{32} - 1) \\ & \wedge 0 \leq \mathit{tag} \leq (2^{6} - 1) \\ & \wedge 0 \leq \mathit{level} \leq (2^{3} - 1) \\ & \wedge 0 \leq \mathit{distinctor} \leq (2^{3} - 1)\end{aligned}$$

Those data fields that have not been explicitly mentioned are boolean flags and use therefore exactly one bit.

3.2 Trace Extraction

Trace data messages are generated in so-called "embedded trace" units. These special hardware units observe the internal states of the SoC and emit compressed runtime information via a dedicated trace port. Amongst the information that an "embedded trace" unit outputs is the information whether a branch has been taken or not. Optionally, this information can be supplemented by the amount of clock cycles (cycle accurate trace) or timestamps.

There are several "embedded trace" implementations available. The most important are Nexus 5001™ [9] based implementations (for instance within the Freescale™ Qorriva/QorIQ [7] devices) and the ARM Coresight™ architecture [3]. The latter consists of different variants of program execution observers. We will explain the solution presented in this paper on the base of the Program Flow Trace (PFT) architecture [2] which is implemented in most ARM Cortex A series processors.

The PFT unit outputs various message types. Most relevant for detailed execution time measurement purposes are the cycle-accurate atom packets (*Atom*), the branch address packets (*Branch*) and the instruction synchronization packets (*I-Sync*). An *Atom* message indicates whether a branch instruction passed or failed its condition code check and outputs an explicit cycle count indicating the number of cycles since the last cycle count output. A *Branch* message indicates a change in the program flow when an exception or a processor security state change occurs or when the CPU executes an indirect branch instruction that passes its condition code check. *I-Sync* messages output periodically the current instruction address and a cycle count.

Traditional trace processing devices record the trace data stream and forward the data to a workstation for decompression and processing. Unfortunately, there is a discrepancy between trace data output bandwidth and trace data processing bandwidth, which is usually several orders of magnitude slower. This results in very short observation periods and long trace data processing times. Consequently, the statistical relevance of the execution time measurements is deteriorated due to the limited observation period.

Our solution does not store the whole received trace data stream for later offline processing. Instead, the trace data is processed online by FPGA logic. This new approach enables an arbitrary observation time while still enabling precise and detailed execution time measurements from "embedded trace" implementations.

The FPGA-based online processing of the trace data stream consists of three processing steps: First, we extract the source specific information (data from the CPU to be observed) from the trace data stream. In a second step we process the message boundaries and detect the periodic *I-Sync* messages. Finally, we distribute the trace data stream segments (between two consecutive *I-Sync* messages) into a multiplicity of parallel operating processing units which reconstruct the program execution. We are starting from the address transmitted by the *I-Sync* message and reconstruct the program execution by processing the subsequent *Branch* and *Atom* messages. For this purpose, we use pre-processed lookup tables which contain the distances (address offsets) between the individual jumps. This instruction reconstruction unit outputs trace point events containing the pre-processed trace point data together with the corresponding cycle counts i.e. timestamps. These events are being processed by the measurements module introduced in 3.3.

Both the online processing of the trace data and the reconstruction of the executed instructions are very resource-consuming, highly parallelized and partly speculative processes which require specialized hardware modules with high-end FPGAs (Xilinx Virtex®-7 / Ultrascale™ series) and large amounts of high performance external memory providing the lookup table content. But the effort is well spent – the system produces a continuous stream of trace point events, the corresponding amount of clock cycles and the associated pre-processed data as a base for basic block WCET analysis within the next processing stage.

3.3 Continuous Context-Sensitive Aggregation

The next processing stage computes the aggregated timing statistics for each basic block.

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

If a basic block is part of nested loops, we only discriminate the iterations of the innermost loop. This is done due to limited storage for the statistics records.

The continuous aggregation stage is split into three parts: the measurement module, the loop module and the statistics module.

We start the description of these modules by introducing some notation. We denote the stream of trace point events E by the sequence $e_0, e_1, \ldots, e_n$, $n \in \mathbb{N}$ of individual events. Each event $e_i = (p_i, t_i)$ consists of a trace point $p_i \in \mathbb{T}$ (see section 3.1) and a timestamp $t_i \in \{0, \ldots, 2^{64} - 1\}$. Moreover, we compute an unique identifier $id_i = \text{T-ID}(p_i)$ for each trace point.

Measurement Module. The measurement module computes the execution time τ of a basic block by taking the timestamp of the corresponding trace point event and subtracting the timestamp of the predecessor event. If no predecessor event has been emitted yet, then $\tau = 0$.

Loop Module. The loop module decides whether the first or the second statistics record should be updated. It uses its internal state machine to interpret the stream of trace point events emitted by the trace extraction stage.

- A state $S = (p, r_0, r_1, \dots, r_k)$ is a stack-like data structure that consists of k rows and a pointer p to the topmost row in use. The value of k is implementation defined.
- A row r is a record that consists of the data fields *valid*, *id* and *index*. The *valid* bit denotes whether the row is already in use. The *id* field identifies a loop or routine. The *index* field is used to decide which statistics record will be updated.
- We call the row to which p points the active row. The special row $(0, 0, 0)$ is denoted by r_{zero}. The initial state $(0, r_{zero}, r_{zero}, \dots, r_{zero})$ is denoted by S_0.

We define two basic state-manipulation operations called "up" and "down". Let S be some state with

$$S = (p, r_0, r_1, \dots, r_{p+m-1}, r_{p+m}, r_{p+m+1}, \dots)$$

then the operation "up" which inserts a record r_{new} m rows above the active row is defined as:

$$S[\uparrow, m, r_{new}] := (p + m, r_0, r_1, \dots, r_{p+m-1}, r_{new}, r_{p+m+1}, \dots)$$

Let S be some state with

$$S = (p, r_0, r_1, \dots, r_{p-m-1}, r_{p-m}, r_{p-m+1}, \dots, r_{p-1}, r_p, r_{p+1}, \dots)$$

then the operation "down" which conditionally inserts a record r_{new} m rows below the active row is defined as:

$$S[\downarrow, m, r_{new}] := (p - m, r_0, r_1, \dots, r_{p-m-1}, f(r_{p-m}, r_{new}), r_{zero}, \dots, r_{zero}, r_{zero}, r_{p+1}, \dots)$$

with the function

$$f(r_{old}, r_{new}) := \begin{cases} r_{old} & r_{old}.valid = 1 \\ r_{new} & otherwise \end{cases}$$

that only returns the new record r_{new} if the old record is not valid.

We can now define the state transition relation of the loop module's state machine. Let S be some state and e_i, e_{i+1} two subsequent trace point events. For the sake of readability, we use three additional propositions:

- $\alpha = call_i \wedge entry_{i+1}$ denotes that a routine call occurred.
- $\beta = exit_i \wedge return_{i+1}$ denotes that a routine's return to its caller happened.
- $\gamma = distinctor_{i+1} \neq distinctor_i$ denotes that the loop body changed.

The state transition operation is then defined as:

$$S[\rhd, e_i, e_{i+1}] := \begin{cases} S[\uparrow, level_{i+1} - level_i, (1, id_{i+1}, 0)] & \neg\alpha \wedge \neg\beta \wedge (level_{i+1} > level_i) \\ S[\downarrow, level_i - level_{i+1}, (1, id_{i+1}, 0)] & \neg\alpha \wedge \neg\beta \wedge (level_{i+1} < level_i) \\ S[\uparrow, 0, (1, id_{i+1}, 0)] & \neg\alpha \wedge \neg\beta \wedge \gamma \wedge (level_{i+1} = level_i) \\ S[\uparrow, 0, (1, id_{i+1}, 1)] & \neg\alpha \wedge \neg\beta \wedge (id_{i+1} = r_p.id) \\ S[\uparrow, level_{i+1} + 1, (1, id_{i+1}, 0)] & \alpha \\ S[\downarrow, level_i + 1, (1, id_{i+1}, 0)] & \beta \\ S & otherwise \end{cases}$$

Statistics Module. The statistics module updates the execution time statistics. The statistics of each basic block b_i are kept in memory organized as records $m_i = (min_{=1}, max_{=1}, total_{=1}, count_{=1}, min_{>1}, max_{>1}, total_{>1}, count_{>1})$ that contains the minimum, maximum and total measured execution time as well as the number of executions separately for the first ($_{=1}$) or additional ($_{>1}$) iterations of the loop that directly surrounds it. The record used for first iterations is also used if a block does not belong to a loop. (The record used for additional iterations is ignored in this case.)

During the initialisation of the statistics module, the memory is initialised such that each record contains the neutral element, i.e. $(+\infty, 0, 0, 0, +\infty, 0, 0, 0)$. During the runtime of the program, when the execution time τ of a basic block has been measured, the corresponding memory record m_i is updated by $min \leftarrow \min(min, \tau)$, $max \leftarrow \max(max, \tau)$, $total \leftarrow total + \tau$ and $count \leftarrow count + 1$. Depending on the *index* of the active row, either the $_{=1}$-part or the $_{>1}$-part is updated.

Example. Consider figure 2. It shows a C program snippet together with the reconstructed CFG of its binary. Moreover, it shows a possible sequence of trace point events and the state of the loop module after a particular event has been processed. We have used the function $\text{T-ID}(p) = p.tag$ to index the basic block statistics. The WCET estimate of our method is 191 cycles. If we apply context-insensitive maximisation, the estimate would be 258 cycles. Our method is thus very precise due to the distinction of loop contexts.

3.4 Post-Processing and Path Analysis

After the testing cycle has been completed, an execution time statistics is stored in memory for each basic block that has been covered by tests. These statistics are downloaded in the first post-processing step and annotated to the CFG's basic blocks. A basic block is marked infeasible if no statistics have been created for it. This information can be used to detect dead code.

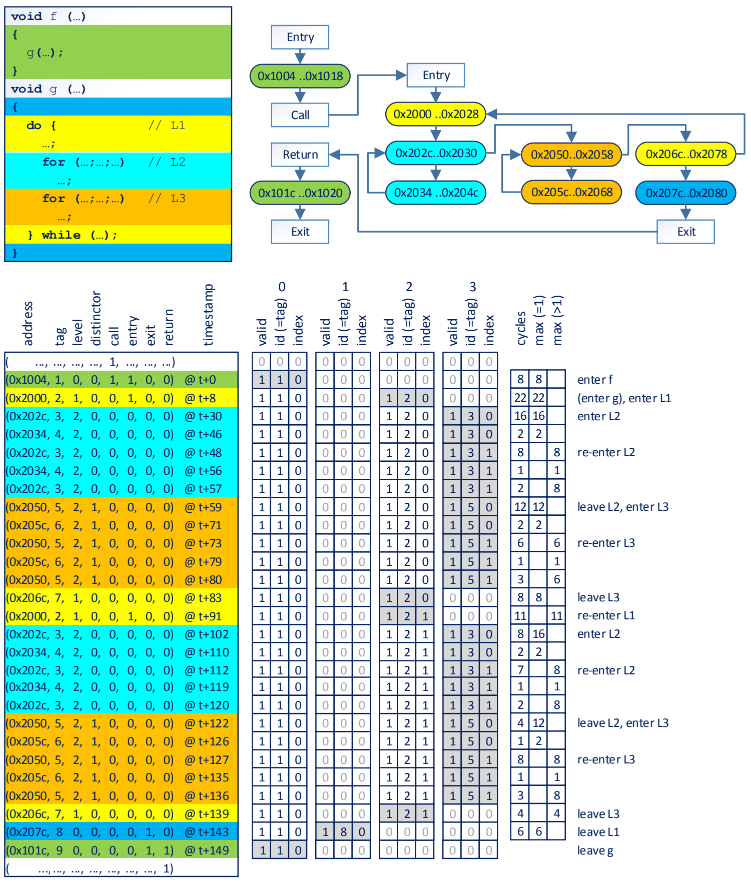
Then, an implicit path enumeration technique is used to find a path with the maximal execution time in the CFG. Our implementation constructs an instance of a maximisation problem expressed via an integer linear program. We refer to [14] for a detailed description of the construction. Afterwards, an ILP solver is used to solve the maximisation instance. Its result is the WCET estimate together with a path in the CFG that induces the estimate.

This path is then used to visualise the WCET contribution of the individual parts of the program. That way, the test engineer can see where in the program the hot spots are. This is particularly useful if the program is the target of performance optimisations.

4 Conclusion and Future Work

In this contribution, we have shown a new hybrid approach to estimate the WCET for modern multicore processors. Our approach uses an FPGA to continuously aggregate the execution time measurements of basic blocks. Moreover, it discriminates between first and further iterations of a loop. This notion of context-sensitivity reflects the typical cache behaviour better than methods that maximise over all executions of a basic block. In our example, the WCET bound is reduced by more than 25%.

Our approach is especially appropriate for those architectures for which a good analytical model cannot be derived, for example due to missing or incomplete documentation. The implementation of the approach is highly scalable and thus applicable to even the fastest processors.



```
void f (…)
{
  g(…);
}
void g (…)
{
  do {              // L1
    …;
    for (…;…;…)     // L2
      …;
    for (…;…;…)     // L3
      …;
  } while (…);
}
```

address	tag	level	distinctor	call	entry	exit	return	timestamp	0 valid	0 id (=tag)	0 index	1 valid	1 id (=tag)	1 index	2 valid	2 id (=tag)	2 index	3 valid	3 id (=tag)	3 index	cycles	max (=1)	max (>1)	
(…	…	…	…	1	…	…	…)		0	0	0	0	0	0	0	0	0	0	0	0				
(0x1004	1	0	0	1	1	0	0)	@ t+0	1	1	0	0	0	0	0	0	0	0	0	0	8	8		enter f
(0x2000	2	1	0	0	1	0	0)	@ t+8	1	1	0	0	0	0	1	2	0	0	0	0	22	22		(enter g), enter L1
(0x202c	3	2	0	0	0	0	0)	@ t+30	1	1	0	0	0	0	1	2	0	1	3	0	16	16		enter L2
(0x2034	4	2	0	0	0	0	0)	@ t+46	1	1	0	0	0	0	1	2	0	1	3	0	2	2		
(0x202c	3	2	0	0	0	0	0)	@ t+48	1	1	0	0	0	0	1	2	0	1	3	1	8		8	re-enter L2
(0x2034	4	2	0	0	0	0	0)	@ t+56	1	1	0	0	0	0	1	2	0	1	3	1	1		1	
(0x202c	3	2	0	0	0	0	0)	@ t+57	1	1	0	0	0	0	1	2	0	1	3	1	2		8	
(0x2050	5	2	1	0	0	0	0)	@ t+59	1	1	0	0	0	0	1	2	0	1	5	0	12	12		leave L2, enter L3
(0x205c	6	2	1	0	0	0	0)	@ t+71	1	1	0	0	0	0	1	2	0	1	5	0	2	2		
(0x2050	5	2	1	0	0	0	0)	@ t+73	1	1	0	0	0	0	1	2	0	1	5	1	6		6	re-enter L3
(0x205c	6	2	1	0	0	0	0)	@ t+79	1	1	0	0	0	0	1	2	0	1	5	1	1		1	
(0x2050	5	2	1	0	0	0	0)	@ t+80	1	1	0	0	0	0	1	2	0	1	5	1	3		6	
(0x206c	7	1	0	0	0	0	0)	@ t+83	1	1	0	0	0	0	1	2	0	0	0	0	8	8		leave L3
(0x2000	2	1	0	0	1	0	0)	@ t+91	1	1	0	0	0	0	1	2	1	0	0	0	11		11	re-enter L1
(0x202c	3	2	0	0	0	0	0)	@ t+102	1	1	0	0	0	0	1	2	1	1	3	0	8	16		enter L2
(0x2034	4	2	0	0	0	0	0)	@ t+110	1	1	0	0	0	0	1	2	1	1	3	0	2	2		
(0x202c	3	2	0	0	0	0	0)	@ t+112	1	1	0	0	0	0	1	2	1	1	3	1	7		8	re-enter L2
(0x2034	4	2	0	0	0	0	0)	@ t+119	1	1	0	0	0	0	1	2	1	1	3	1	1		1	
(0x202c	3	2	0	0	0	0	0)	@ t+120	1	1	0	0	0	0	1	2	1	1	3	1	2		8	
(0x2050	5	2	1	0	0	0	0)	@ t+122	1	1	0	0	0	0	1	2	1	1	5	0	4	12		leave L2, enter L3
(0x205c	6	2	1	0	0	0	0)	@ t+126	1	1	0	0	0	0	1	2	1	1	5	0	1	2		
(0x2050	5	2	1	0	0	0	0)	@ t+127	1	1	0	0	0	0	1	2	1	1	5	1	8		8	re-enter L3
(0x205c	6	2	1	0	0	0	0)	@ t+135	1	1	0	0	0	0	1	2	1	1	5	1	1		1	
(0x2050	5	2	1	0	0	0	0)	@ t+136	1	1	0	0	0	0	1	2	1	1	5	1	3		8	
(0x206c	7	1	0	0	0	0	0)	@ t+139	1	1	0	0	0	0	1	2	1	0	0	0	4		4	leave L3
(0x207c	8	0	0	0	0	1	0)	@ t+143	1	1	0	1	8	0	0	0	0	0	0	0	6	6		leave L1
(0x101c	9	0	0	0	0	1	1)	@ t+149	1	1	0	0	0	0	0	0	0	0	0	0				leave g
(…	…	…	…	…	…	…	1)																	

Figure 2 Exemplary application of our method. The figure shows a C program snippet (left upper corner), the reconstructed CFG (right upper corner), a sequence of trace point events (bottom left corner), the states of the loop module (bottom middle) and the continuous aggregation of the maximum execution time (bottom right corner). Basic blocks are shown with the address of their first and last instruction. Colors have been used to show which event belongs to which basic block. The active row is highlighted in grey.

We are currently building a prototype hardware implementation as part of the research project CONIRAS. As soon as the prototype is completed, our approach will be validated with some typical embedded real-time applications, e.g. the set of WCET benchmarks [8].

The statistics computed by our method are rather simple. The question whether more advanced ones could be performed online as well needs further investigation.

Acknowledgements. The authors like to thank Christian Ferdinand, Michael Schmidt and the anonymous reviewers for their valuable comments.

References

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

2 ARM Ltd. CoreSight™ program flow trace™ PFTv1.0 and PFTv1.1 architecture specification, 2011.

3 ARM Ltd. CoreSight™ architecture specification v2.0 ARM IHI 0029b, 2013.

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

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

6 J.-F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In R. Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.

7 Freescale Semiconductor, Inc. P4080 advanced QorIQ debug and performance monitoring reference manual, rev. f, 2012.

8 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

9 IEEE-ISTO. IEEE-ISTO 5001™-2003, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface, 2003.

10 R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th Euromicro International Workshop on WCET Analysis*, pages 67–70, June 2004.

11 J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multicore Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *ECRTS'14: Proceedings of the 26th Euromicro Conference on Real-Time Systems*, July 2014.

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

13 S. Stattelmann and F. Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

14 H. Theiling. ILP-based Interprocedural Path Analysis. In A. L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of EMSOFT 2002, Second International Conference on Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2002.

15 H. Theiling. *Control Flow Graphs for Real-Time System Analysis. Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis*. PhD thesis, Saarland University, 2003.

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

17 N. Williams. WCET measurement using modified path testing. In R. Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.

Context-sensitive Parametric WCET Analysis

Clément Ballabriga, Julien Forget, and Giuseppe Lipari

Université Lille, CRIStAL, UMR 9189, 59650 Villeneuve d'Ascq, France
`{Clement.Ballabriga,Julien.Forget,Giuseppe.Lipari}@univ-lille1.fr`

Abstract

In this paper, we propose a WCET analysis that focuses on two aspects. First, it supports context-sensitive hardware and software timing effects, meaning that it is sensitive to the execution history of the program and thus can account for effects like cache persistence, triangular loop, etc. Second, it supports the introduction of parameters in both the software model (e.g. parametric loop bounds) and the hardware model (e.g. number of cache misses). WCET computation by static analysis is traditionally handled by the Implicit Path Enumeration Technique (IPET), using an Integer Linear Program (ILP) that is difficult to resolve parametrically. We suggest an alternative tree-based approach. We define a context-sensitive CFG format to express these effects, and we provide an efficient method to process it, giving a parametric WCET formula. Experimental results show that this new method is significantly faster and more accurate than existing parametric approaches.

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

Keywords and phrases Parametric, WCET, Real-time, Static analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.55

1 Introduction

In static WCET analysis methods, an upper bound to the WCET of a task is traditionally computed in three steps. First, the task code is statically analyzed to model the set of possible execution paths. Then, the hardware is taken into account by modeling the architectural effects: local effects (timings of basic blocks) and global effects (interactions between basic blocks). Finally, the WCET computation takes as input a program and its environment (hardware and software), and produces the WCET. A popular technique for doing this last step is IPET [14], in which the WCET computation is represented as an ILP problem solving.

With traditional WCET computation, if the program, input values, software environment or hardware platform changes, it is necessary to re-run the entire analysis. On the opposite, *parametric* WCET analysis takes a parametrized input, and produces a formula that depends on those parameters. If the parametric values change, it is possible to compute a new WCET simply by evaluating the formula on the new values. This offers several benefits, which we detail below.

First, parametric WCET avoids re-running the entire WCET computation each time there is a minor change to the program or hardware configuration. This is an important aspect, due to the increasing size of real-time systems and to the non-linear complexity of WCET analyses. Similarly, parametric WCET simplifies the analysis process when third-party software is involved, since the developer can provide a parametric WCET that can be adapted to the target system (software and hardware).

Second, many system parameters are only known at run-time: loop bounds that depend on input values, software and hardware state changes, operating system interference, etc. Using a parametric WCET, it is possible to evaluate the WCET formula at run-time and take better decisions accordingly. For instance, tighter WCET evaluation at run-time could benefit energy-aware scheduling techniques based on Dynamic Voltage and Frequency Scaling (DVFS) [12].

Finally, large execution time values may happen only very rarely, for instance for unlikely combinations of input data. By using parametric WCET, it is possible to design the system according to an upper bound that is safe for the vast majority of executions of the system, and then evaluate a parametric WCET formula at run-time to trigger an alternate less time-consuming computation when the formula returns a value exceeding the safe bound (and thus remain under the safe bound).

In this work we present a novel approach to parametric WCET analysis. Unlike the majority of existing approaches, our methodology is not based on ILP. To the best of our knowledge, it is the first to benefit from (1) a reasonable computing time, since it runs in polynomial time, and (2) a good precision thanks to the support of *context-sensitive effects* (persistent cache blocks, non-rectangular loops, branch prediction, etc.). Furthermore, in our approach the trade-off between computation time and precision is configurable: it is possible to increase (or decrease) the context sensitivity to improve the precision (or speed up the computation time).

2 Related works

In [1], a technique is presented to perform a partial, composable WCET analysis. This work addresses mostly the software and hardware modeling that occurs before the WCET computation proper. Results are presented for the instruction cache and branch prediction analysis, and loop bounds estimation. However, no solution is provided to perform the ILP computation parametrically.

Feautrier [8] presented a method for parametric ILP computation. A traditional ILP solver takes an ILP system as input, and provides a numerical solution corresponding to the maximization (or minimization) of an objective function. The ILP solver presented in [8] (called *PIPLib*), takes a parametrized ILP system as input, and produces a *quast* (quasi-affine selection tree). Once computed, this tree can be evaluated for any valid parameter values, without having to re-run the solver. However, this approach is computationally very expensive. In contrast, our approach has polynomial complexity, and it is therefore scalable, whereas the introduced pessimism is very small, as it will be shown in Section 6.

In theory, PIPLib could be applied for solving parametrically the ILP systems produced in the context of the IPET method [14]. However, experimentations [4] have shown that it does not scale well: the parametric ILP solving may become intractable for medium to large size programs. The MPA (Minimum Propagation Algorithm) [5, 4] attempts to address the shortcomings of PIPLib in the context of the IPET method. MPA takes as input the results of the software and hardware modeling analysis, and produces directly a parametric WCET formula. Compared with MPA, our method is significantly tighter because it takes into account various context-sensitive software and hardware timing effects.

In the past, many tree-based WCET computation methods have been presented [11]. In [7] the authors suggest a method to compute the WCET parametrically using a tree-based approach. Our parametric approach is also tree-based, but unlike the one presented in [7], it can work directly on the binary target (no source code needed). Furthermore, our method can model timing effects in a more generic and accurate way than existing tree-based approaches, and the trade-off between accuracy and computation time can be configured.

ParaScale [12] is an approach to exploit variability in execution time to save energy. By statically analyzing the tasks, a parametric WCET formula is given for loops in terms of the loop iteration count. At run-time, before entering a loop, the formula is evaluated and the system dynamically scales the voltage and frequency of the processor. In comparison, our parametrization is not limited to loop bounds, but can be used for anything influencing the WCET (such as cache misses, branch predictor states, etc.). Furthermore our approach uses a more refined model for loops.

Finally, note that our method provides an alternative to the time-consuming ILP solving, thus our method is competitive even compared to non-parametric WCET analysis based on ILP.

3 Context-sensitive model

Our algorithm takes as input a *context-sensitive Control Flow Graph* (CFG) and produces as output a parametric formula. Let us motivate the need for the context-sensitive CFG as our algorithm input by using two examples.

First we consider the instruction cache analysis by categorization. In this approach, blocks can be categorized as *persistent* with respect to a loop (for the sake of simplicity, we assume that each basic block matches exactly a cache block), meaning that the block will stay in the cache during the whole execution of the loop (only the first execution results in a cache miss). In the CFG shown in Figure 1(a), if we assume that Block 3 is persistent, its execution time depends on whether it has already been executed or not. With IPET-based approaches, this information would be stored as an ILP constraint, but since we do not use IPET, we need to store it in the CFG itself.

As a second example, let us consider a triangular loop: a *for* loop $i = 1..10$, containing an inner *for* loop $j = i..10$. The maximum iteration count for each loop is 10, but the inner loop body can be executed at most $\sum_{i=1}^{10} i$ times. Once again, this cannot be expressed with the traditional CFG. In both examples, there is a block of code whose execution time depends on the number of times this block was executed after entering some loop containing it: this is what we call the *execution context*.

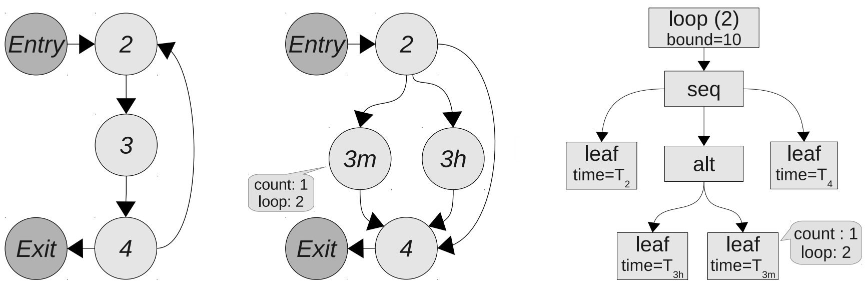
3.1 Context-sensitive CFG

In IPET-based WCET analyses, the various preliminary analyses (such as cache block categorization) produce ILP constraints. In our approach, instead we transform the program CFG into a *context-sensitive* CFG (and possibly make other graph transformations such as block duplication).

The context-sensitive CFG extends the standard CFG by allowing an (optional) context annotation associated with each node. For any CFG G, let L_G denote the set of its loops. A loop is defined as the set of cycles in the CFG sharing the same loop header. A loop header is defined as a node n having a predecessor p such that n dominates p. For any header h, we will note L_h the loop having the header h, and for any node n, we will note L_n the loop immediately containing n (i.e. L_n is the loop containing n such that there is no other loop containing n, and whose header is inside L_n). Furthermore, let us define a partial order on loops: $\forall (L_1, L_2) \in L_G \times L_G, L_2 < L_1$ if (and only if) L_2 is contained in L_1 (i.e. if there exists a path from the header of L_1 to itself, going through the header of L_2).

A context-sensitive CFG G is defined by $G =< \mathcal{B}_G, \mathcal{E}_G, \mathcal{A}_G >$, where $\mathcal{B}_G$ is the set of nodes (basic blocks), $\mathcal{E}_G$ is the set of edges, and $\mathcal{A}_G = L_G \times \mathcal{B}_G \times \mathbb{N}$ is the set of context annotations. We will note T_b the execution time of basic block b.

A context annotation $a = (a_l, a_b, a_n)$ represents a restriction on the set of feasible paths in the CFG: the basic block a_b (inside a_l) may be executed at most a_n times each time the loop a_l is entered.



(a) Standard CFG. **(b)** Context-sensitive CFG. **(c)** Expression tree.

Figure 1 Program representations.

To model our first example (cache persistence), we split Block 3 from Figure 1(a) as two (virtual) blocks $3h$ and $3m$, representing respectively the hit and miss. Then we add a context annotation to Block $3m$, as shown in Figure 1(b). This represents the fact that $3m$ can be executed only once per execution of the loop. To model the triangular loop example, we can add Annotation $(L_{outer}, body, \sum_{j=1}^{10} j)$, where L_{outer} represents the outer loop, and *body* is the block inside the inner loop.

These annotations are intended to be a generic tool to model many WCET-related effects (hardware, and software), therefore the exact way to generate those annotations will depend on the effect we want to model (and on the underlying analysis). However, the algorithm evaluating the context-sensitive CFG is generic and does not need to know which types of effects are represented by the annotations.

If a CFG node contains no annotation, its execution time is context insensitive (i.e. it is the same for all contexts). In this trivial case, its WCET is an integer. Otherwise, its execution time depends on the context. We define the context of a node n as the history of *context events* happening before the execution of n. The set of context events is defined as $V = \{exec(n) | n \in \mathcal{B}_G\} \cup \{loop(l) | l \in L_G\}$, where $exec(n)$ represents the execution of node n, and $loop(l)$ represents the entrance in loop l containing (not necessarily directly) n. Then, a context is defined as a list of context events. We denote an empty context as ϵ. We also use the classical list notation $\mathsf{head} \cdot \mathsf{tail}$ to denote a context whose first element is head, followed by list tail. For any context c and any integer k we note c^k the repetition k times of sequence c. We note $range[L_1, L_n]$ the sequence $loop(L_1) \cdot ... \cdot loop(L_n)$ where $\forall i, L_1 < L_i < L_n$. Similarly, the notation $range[L_1, L_n[$ corresponds to $loop(L_1) \cdot ... \cdot loop(L_{n-1})$. Finally, the set of contexts is noted C. For instance, the context of the third execution of the block $3m$ (in loop body) in Figure 1(a) would be $loop(L_2) \cdot exec(3m) \cdot exec(3m)$ (the loop represented in the figure is noted L_2, as its header is the basic block 2).

3.2 Expression tree

The execution time of a node depends on the node itself and on the context. Therefore, it is possible to compute an unique execution time for any $(context, node)$ pair. To provide a practical and easily computable approach, we will first convert the context-sensitive CFG into an *expression tree*, then evaluate this tree to produce the WCET. To perform this conversion, an algorithm similar to the one presented in [6] can be used. An expression contains a set $\mathcal{T}$ of tree nodes which can be of type *alt*, *seq*, *loop* and *leaf*. A leaf node n represents a single basic block, and has an attribute storing the basic block execution time (noted n_{time}). A seq node represents a sequence of child nodes, and an alt node represents a choice (alternative) between child nodes. A loop node represents a loop in the program, has exactly one child node (representing the loop body), and has an attribute (noted n_b) representing the maximum iteration count. In addition, each context annotation in the original CFG is copied to the corresponding leaf node in the tree[1]. Figure 1(c) shows the tree that would be generated from the context-sensitive CFG represented in Figure 1(b).

3.3 Abstract WCET values

An abstract WCET represents a set of possible execution times for a tree node, along with conditions required for that execution time to occur. In our cache example, the abstract WCET value computed for the alt node would contain two execution times (miss case, and hit case) and indicate that the miss case can occur only the first time Block 3 is accessed.

First, we introduce the concept of *context mapping*. A context mapping $m = (m_{loop}, m_{time})$ is a pair whose first element (m_{loop}) represents a loop, and second element (m_{time}) represents an execution time. The set of context mappings is noted $M = L_G \times \mathbb{N}$. We define a total non-strict order on M such that $\forall (m, n) \in M \times M, m \geq n \iff m_{time} \geq n_{time}$. An abstract WCET $\alpha = (\alpha_{map}, \alpha_{other})$ is a pair, in which the first element (α_{map}) is a multiset over M, and the second element (α_{other}) is an execution time. The set of abstract WCETs is noted $A = M^{\#} \times \mathbb{N}$, where $M^{\#}$ is the sets of multisets over M. For any multiset m over M, we note $1_m : M \to \mathbb{N}$ its multiplicity function. Furthermore, we define the $\max_k : M^{\#} \to M^{\#}$ function, returning the k greatest elements of $M^{\#}$ (context mappings with same time but different loop are considered equivalent by this function).

An abstract WCET α is computed for each node in the tree. Informally, the presence of a context mapping m in α_{map} means that the node may have an execution time of m_{time}, but only once each time m_{loop} is entered. The α_{other} value represents the default execution time of the node (used whenever no other time can be used due to context). We lift the concept of context over CFG nodes to tree nodes. The new set of contexts is defined as $V' = \{exec(n) | n \in \mathcal{T}\} \cup \{loop(l) | l \in L_G\}$.

The abstract WCET of a tree node n provides a mapping from contexts to execution times. We define two functions, $eval : C \times A \times \mathcal{T} \to \mathbb{N}$, and $next : C \times A \times \mathcal{T} \to \mathbb{N}$. Let n be a tree node, α its associated abstract WCET, and c a context for the tree node n. The expression $eval(c, \alpha, n)$ gives the WCET corresponding to the execution of the event sequence (loop entrances, and executions of n) represented by c, while $next(c, \alpha, n)$ gives the WCET increase caused by a subsequent execution of n after the event sequence represented by c (which is not necessarily equal to the WCET of this last execution of n). The function $next$ is defined such that $\forall \alpha, next(c, \alpha, n) = eval(c \cdot exec(n), \alpha, n) - eval(c, \alpha, n)$, and $eval$ is defined as follows:

$$eval(c, \alpha, n) = \alpha_{other} \times \max(0, i - |\Theta|) + \sum_{t \in \max_n(\Theta)} t$$

where i is the number of $exec(n)$ present in c, and Θ is such that $\forall m \in M, 1_\Theta(m) = 1_{\alpha_{map}}(m) \times |\{k | c_k = loop(m_{loop})\}|$. The general idea behind the $eval$ function is to account for the execution time of n for each $exec(n)$ present in the context, while ensuring that each context mapping is not used more times than the number of times its corresponding loop is entered, and using α_{other} to provide a time for n when no context mapping can be

[1] It is possible to extend the CFG annotations to sub-graphs representing if or loop structures (as opposed to single blocks), this is not described here for the sake of brevity.

used. The *eval* function may compute an over-approximation for some contexts, however this representation is quite compact, with many contexts (leading to the same execution time) described by a single context mapping value.

4 WCET computation

Let us define some notations that we will use in this section. The binary operator $\uplus$ is defined such that for any multiset pair (m, n), $\forall x, 1_{m \uplus n}(x) = 1_m(x) + 1_n(x)$. Similarly to the set-builder notation, we define the multiset-builder notation $[n|n \in m \wedge pred(n)]$, representing the multiset containing all the elements in m satisfying $pred(n)$. For any element e in a multiset, and any $n \in \mathbb{N}$, we will note $e \otimes n$ the multiset containing only the element e, with a multiplicity of n.

4.1 Node evaluation

We note ω the evaluation function, taking as input a tree node and producing an abstract WCET. Once the tree evaluation is finished, and we have the abstract WCET corresponding to the root node *root*, the concrete WCET is $\omega(root)_{other}$. We detail below a simple way to compute function ω.

For a leaf node n, if the node has no context annotation then $\omega(n) = (\emptyset, n_{time})$, otherwise, $\omega(n) = (\{(a_l, n_{time})\} \otimes a_n, 0)$, where $(a_l, a_b, a_n) \in A$ is the context annotation associated with the leaf node (a_b is the basic block represented by leaf node n).

For a seq node n with two children n_1, n_2 (this can be extended to an arbitrary number of children, since the operation is associative), we have $\omega(n) = (map, other)$ such that:

$$other = \omega(n_1)_{other} + \omega(n_2)_{other} \quad map = \biguplus_{l \geq L_n} \mathcal{S}(\omega(n_1), \omega(n_2), range[l; L_n])$$
$$\text{with } \mathcal{S}(\alpha, \alpha', c) = \biguplus_{i=0}^{m-1} (l, next(c \cdot exec(n_1)^i, \alpha, n_1) + next(c \cdot exec(n_2)^i, \alpha', n_2))$$
$$\text{and } m = \max(|\alpha_{map}|, |\alpha'_{map}|)\,.$$

The general idea is to match context mappings from both children, and add their times. For example, if $\omega(n_1) = ([(L_1, 10), (L_1, 8)], 5)$ and $\omega(n_2) = ([(L_2, 4)], 3)$, and $L_2 < L_1$, then $\omega(n) = ([(L_1, 10 + 4), (L_1, 8 + 3), (L_2, 5 + 4)], 5 + 3)$.

For a alt node n with two children n_1, n_2 (this can be generalized in the same way as the seq node), $\omega(n) = (map, other)$, such that:

$$other = \max(\omega(n_1)_{other}, \omega(n_2)_{other})$$
$$map = [m|m \in (\omega(n_1)_{map} \uplus \omega(n_2)_{map}) \wedge m_{time} > other]$$

For any loop node n representing loop l, with body n_1, $\omega(n) = (map, other)$ such that:

$$other = eval(\sigma(l, n_1, n_b), \omega(n_1), n_1) \quad map = \biguplus_{l' > l} \biguplus_{i=1}^{j} (l', \tau(i, n_1, n_b, range[l'; l[))$$
$$\text{with } \tau(i, n, b, c) = eval(c \cdot \sigma(l, n, b)^i, \omega(n), n) - eval(c \cdot \sigma(l, n, b)^{n-1}, \omega(n), n)$$
$$\text{and } \sigma(l, n, k) = loop(l) \cdot exec(n)^k, \quad j = min(\{i|\tau(i, \alpha, n_1, b, c) \leq other\})\,.$$

For example, if we consider a loop l having a body with abstract WCET $\omega(n_1) = ([(L_1, 8),$ $(L_1, 7)], 6)$ and loop bound 5, inside an outer loop L_1, then $\omega(n) = ([(L_1, 8 + 7 + 6 \times 3)], 6 \times 5)$.

The abstract WCET value corresponding to the seq node of Figure 1(c) is $([(L_2, T_2 + T_{3m} + T_4)], T_2 + T_{3h} + T_4)$: the first time the seq node is executed after entering loop 2, its WCET value is $T_2 + T_{3m} + T_4$, however for subsequent executions it is $T_2 + T_{3h} + T_4$. The abstract WCET value corresponding to the loop node is $(\emptyset, 10 \times (T_2 + T_4) + 9 \times T_{3h} + T_{3_m})$: each time we enter the loop, a miss will occur at most once (Block $3m$) and other iterations will be hits (Block $3h$). Note that, since there is no context mapping in the abstract WCET (the multiset is empty), it is equivalent to a static WCET.

4.2 Approximations

In the presence of many context annotations, tree nodes can have many possible execution times, thus the evaluation can produce very large context mappings. Using measurement results from various experiments, we have observed that most of the time, the function that maps iteration counts to execution times can be tightly over-approximated by a much simpler piecewise linear function. The presence of a straight-line section in this piecewise linear function means that there is a group of context mappings in the corresponding abstract WCET value, with a time approximately equal to the slope of the straight-line section. Such a group of i context mappings can be merged into one context mapping with a multiplicity of i, and a time equal to the maximum time of the former group.

Using such an approximation, we lose some precision but we reduce the amount of data we will have to process. We can only merge context mappings referring to the same loop. Therefore, in order to increase the merging possibilities, for any context mapping $(L_1, t) \in M$, it is possible to replace it safely by (L_2, t), with $L_2 < L_1$. Of course this would cause a loss of precision, but it allows more merging and reduce the complexity of the evaluation.

The greater the number of context mappings we use, the greater the precision and the analysis time. Therefore, when evaluating the tree, at each node we may need to use heuristics to decide when and how to perform an approximation. For the experiments performed in this paper, we use a simple (yet quite effective) strategy: we perform an approximation whenever the number of distinct context mappings exceeds a certain (user-configurable) threshold.

5 Parametric WCET computation

Our tree-based computation is the first step for parametric computation. It can be made parametric in a much easier way than an ILP computation (while it is true that ILP solving can be done parametrically, it is way more costly than our method), as we will show in this section. For instance, considering the example of Figure 2 and assuming that the loop bound is not known statically, we will show how to create a parametric WCET in terms of the loop bound value and how to obtain the concrete WCET once the loop bound value is known.

We extend our expression tree model, to enable the introduction of parameters that represent information unknown at static analysis time:

- We introduce a new node type: a *param* node n has an attribute n_{param} representing a parameter identifier. Such a node can represent any type of (statically unknown) expression sub-tree. It can be used to perform modular analysis (to represent the abstract WCET for a separately-analyzed library call for which we do not have the code).
- In loop nodes, the maximum iteration count, which was previously stored as an integer, can now be a parameter identifier, to represent a statically unknown loop bound.
- In any context annotation (a_l, a_b, a_n) associated to a node, the values a_l and a_n can now be parameter identifiers. This could be used to support parametric cache categories.

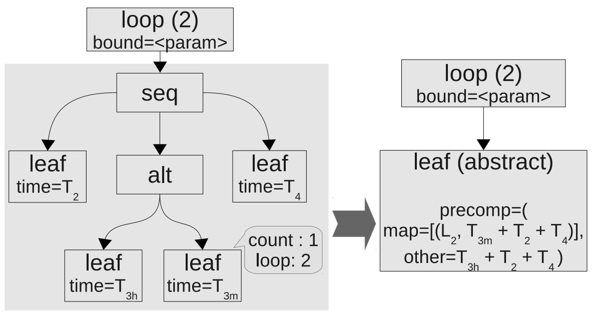


Figure 2 Partial evaluation.

5.1 Partial evaluation

The parametric WCET computation starts with a partial evaluation phase, which precomputes as much as possible from the parametric tree, and produces a simplified parametric tree.

We introduce two separated kinds of leaf nodes: a leaf node n can either be concrete (associated with a basic block), or abstract (with an attribute $n_{precomp}$ of type A, abstract WCET). The abstract leaf node holds the result of a precomputed sub-tree. The evaluation function ω for an abstract leaf node is defined such that $\omega(n) = n_{precomp}$.

A node is *parametric* if it (or any of its descendants) contains a parameter. To partially evaluate the parametric tree, we select any non-parametric non-abstract node n, remove it (and its descendants), and replace it by an abstract leaf node n' such that $n'_{precomp} = \omega(n)$. This is repeated until the tree contains only parametric nodes, and abstract leaf nodes. Some optimizations can also be applied (not detailed here) to remove unneeded nodes.

In our example, the result of the partial evaluation is shown in the right side of Figure 2, containing only one parametric loop node, and an abstract leaf node holding the abstract WCET corresponding to the loop body: the first time the loop body is executed, its time is $T_{3m} + T_2 + T_4$, subsequently its time is $T_{3h} + T_2 + T_4$.

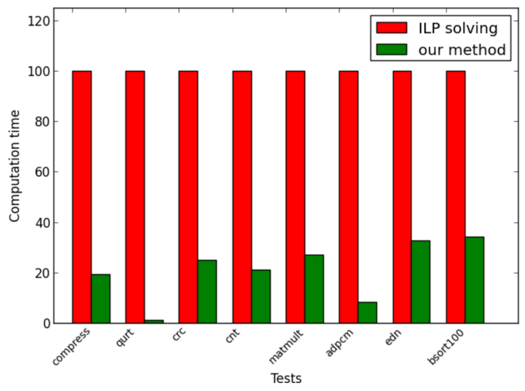
5.2 Parameter instantiation

The next step is the parameter instantiation, which takes as input a simplified parametric tree and parameter values. To do the parametric instantiation, we replace, in the parametric tree, each param node n by the value of n_{param}, and each parameter p present in a loop bound or in a context annotation by their value. Once the resulting tree contains no parametric node, it can be evaluated using the method from Section 4, producing the WCET.

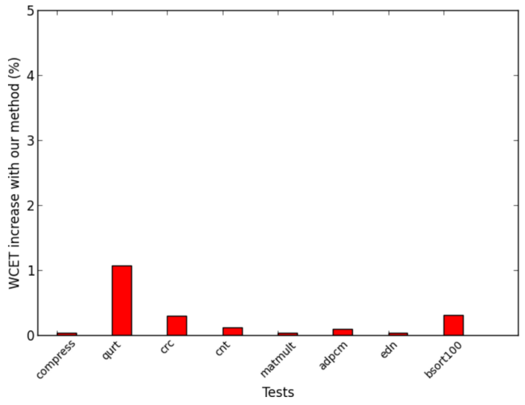
In our example, if we want to instantiate the simplified tree with a loop bound of 10 for L_2, we replace $< param >$ with 10. Let us call r this partial result after replacement. We can evaluate it, giving the abstract WCET $\omega(r)$, from which we can get the concrete WCET: $\omega(r)_{other} = T_{3m} + T_{3h} \times 9 + 10 \times (T_2 + T_4)$.

6 Experiments

To evaluate the performance of our approach, we compared it to existing IPET-based WCET analysis. The target hardware is an ARM processor with a set-associative LRU instruction cache (the data cache is not taken into account). The processor pipeline is analyzed with the exegraph method [13] and the instruction cache is modeled using cache categorization [9]. We perform our analysis on a subset of the Mälardalen benchmarks [10], used as standalone



(a) Analysis time comparison.



(b) Pessimism measurement.

Figure 3 Experimental results.

tasks, without any modeling of the operating system. To perform the preliminary steps of the WCET analysis (program path analysis, CFG building, loop bounds estimation, pipeline and cache modeling), we used OTAWA, an open source WCET computation tool [2]. Then, we compare our approach with an ILP approach (using the GNU lp_solve ILP solver [3]), by running both on the result produced by OTAWA.

We first compare the ILP solving time and the time taken by our tree evaluation (Figure 3(a)). The times are normalized so that the ILP time is always 100. The measurements do not include the preliminary WCET analyses (performed by OTAWA) as they are common to both approaches. On average, our approach reduces the analysis time by a factor of 15.7. The running time of our approach is polynomial in the tree size as long as we merge context mapping groups once their size exceeds a fixed threshold.

Our approach uses simplifications that discard information, which introduces pessimism. To quantify it, we choose a large loop in each benchmark, and create a parameter representing its iteration count. We run our parametric analysis on each benchmark, instantiate the result for each possible parameter value, and we evaluate the pessimism by comparing the result to the value obtained by IPET. Figure 3(b) shows the average WCET increase for each benchmark. We can see that the pessimism increase is very reasonable (on average 0.25%).

The pessimism can be attributed to two main causes: (1) the reduced expressiveness of our annotated CFG format (as opposed to an ILP system), and (2) the approximations performed during our computation (such as the simplification by merging context mappings presented in Section 4.2). We cannot express all types of infeasible paths, such as mutually exclusive paths, although we plan to support this in the future.

7 Conclusion

In this paper, we have presented a new approach to perform the final step of WCET computation, which replaces the ILP solving phase of the IPET method. Instead of computing a single, fixed WCET value, our method computes a WCET formula that may depend on parameters, such as, for instance, loop bounds, architectural entry states (cache or branch predictor state), or system environment (for example, preemption count). The WCET formula can be evaluated once the parameter values are known. Additionally, it has a significantly faster computation time, even compared to non-parametric WCET computation methods, while retaining good precision, and supports various WCET features such as branch prediction, cache analysis, and non-rectangular loops.

The main limitation of our method is the diminished expressiveness (compared to ILP approaches), for instance in presence of certain types of infeasible paths. In future works, we plan to work on this issue, and enable our context-sensitive CFG format to represent all characteristics found in modern WCET analyses.

References

1 Clément Ballabriga, Hugues Cassé, and Marianne De Michiel. A Generic Framework for Blackbox Components in WCET Computation. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2009.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*. Springer, 2010.

3 Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. `http://lpsolve.sourceforge.net/5.5/`.

4 S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.

5 Stefan Bygde. *Parametric WCET Analysis*. PhD thesis, Mälardalen University Press, 2013.

6 Cristina Cifuentes. A structuring algorithm for decompilation. In *XIX Conferencia Latinoamericana de Informática*, 1993.

7 Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *14th Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2002. IEEE Computer Society.

8 Paul Feautrier. Parametric integer programming. *RAIRO Operations Research*, 22, 1988.

9 Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

10 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks – past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

11 Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. In *Real-Time Systems Symposium (RTSS)*, pages 97–108, 1995.

12 S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *Real-Time Systems Symposium (RTSS)*, 2005.

13 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. In *Transactions on High-Performance Embedded Architectures and Compilers II*, volume 5470 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

14 Yau tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Real-Time Systems Symposium (RTSS)*, 1995.

WCET and Mixed-Criticality: What does Confidence in WCET Estimations Depend Upon?*

Sebastian Altmeyer[1], Björn Lisper[2], Claire Maiza[3], Jan Reineke[4], and Christine Rochange[5]

1 **University of Luxembourg, Luxembourg**
`sebastian.altmeyer@uni.lu`
2 **Mälardalen University, Sweden**
`bjorn.lisper@mdh.se`
3 **Université Grenoble Alpes, Verimag, France**
`claire.maiza@imag.fr`
4 **Saarland University, Germany**
`reineke@cs.uni-saarland.de`
5 **University of Toulouse, France**
`rochange@irit.fr`

Abstract

Mixed-criticality systems integrate components of different criticality. Different criticality levels require different levels of confidence in the correct behavior of a component. One aspect of correctness is timing.

Confidence in worst-case execution time (WCET) estimates depends on the process by which they have been obtained. A somewhat naive view is that static WCET analyses determines safe bounds in which we can have absolute confidence, while measurement-based approaches are inherently unreliable. In this paper, we refine this view by exploring sources of doubt in the correctness of both static and measurement-based WCET analysis.

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

Keywords and phrases mixed criticality, WCET analysis, confidence in WCET estimates

Digital Object Identifier 10.4230/OASIcs.WCET.2015.65

1 Introduction

Due the integration of multiple safety levels (A to E in DO178B for certification in avionics, or A to D in ISO 26262, a functional safety standard for automotive), tasks of different criticality may be executed on a shared platform. A naive approach to certifying such a *mixed-criticality* system is to apply the certification methods corresponding to the highest present level of criticality to *all* tasks. The drawback of this approach is that low-criticality tasks become unnecessarily costly to validate and the system analysis potentially pessimistic.

Research in mixed-criticality scheduling targets the validation of these system assuming the certification requirements are reflected in the execution time bounds, which vary depending on the associated criticality. The original model by Vestal [35] suggests two levels of timing estimations:

* This work was partially supported by French ANR under grant ANR-12-INSE-0001, the German Science Foundation (DFG) as part of the Transregional Collaborative Research Center SFB/TR 14 (AVACS) and by the COST Action IC1202: Timing Analysis On Code-Level (TACLe).

- C(HI) is a high-confidence WCET estimate, and
- C(LO) is a lower confidence estimate of the WCET,

but there may be as many WCET estimations as safety levels.

The rationale behind this model is to guarantee schedulability of all tasks with their C(LO) bounds, while ensuring that if a task exceeds its low criticality execution time bound, the highly-critical tasks are able complete their execution within their deadlines as long as their execution times do not exceed their C(HI) bounds. Intuitively, C(LO) $\leq$ C(HI) as it is assumed in Vestal's model. However, a more reliable WCET analysis provides higher confidence in the validity of its estimation, but does not necessarily result in a greater bound. Also, increasing the effort to analyse a task does not necessarily increase the bound, but may even reduce it.

In this paper, we present our point of view on the sources of these different levels of confidence in the WCET estimations. Our aim is to discuss the sources of doubt in the correctness of WCET estimations. In contrast to [10], where confidence in WCET estimations and monotonicity of WCET estimations with respect to the different certification levels have already been shortly discussed, we highlight the problem from the perspective of timing analysis. We first discuss confidence in static and measurement-based WCET analysis methods, then focus on the impact of multi-core platforms on these sources of doubts and specifically on the interferences on shared resources. We conclude with a discussion of open problems.

Please note that we restrict our attention in this paper to deterministic approaches currently used in industry (due to limited space). As future work, we plan to enlarge our study to recently introduced probabilistic approaches.

2 On Confidence in Static WCET Analysis

2.1 Structure of Static WCET Analysis Tools

Static timing analyses compute bounds on a task's execution time by analysing the task characteristics and determining its behaviour on the target machine statically, i.e, without executing the task on the target platform. The techniques employed by static timing analyses, such as abstract interpretation or model checking, are borrowed from the related fields compiler construction and verification and are meant to be sound by construction.

A static timing analysis typically consists of three phases, ISA-level analysis, microarchitectural analysis and path analysis. ISA-level analysis derives flow information of the task, such as loop bounds, effective memory addresses of memory read or write and pointer addresses. Microarchitectural analysis derives bounds on the execution times of each basic block. Path analysis combines the information of the previous steps and determines the longest execution path through the program.

Microarchitectural analysis has to resort to the level of the binary as only on this level complete information about the task behaviour is available. ISA-level analysis can operate on both, the high-level and on the binary, yet requires a control-flow graph representation of the program. Consequently, additional steps are needed to bridge the gap between the different representations and to establish a connection between the main phases of the analysis.

2.2 ISA-level Analysis

Some important supporting analyses depend only on the untimed, "functional" semantics of the code. This includes *program flow analysis*, which attempts to find program flow constraints such as loop bounds, or infeasible path constraints. Also a conventional *value analysis* is often needed, for purposes like bounding the possible addresses for memory accesses.

There are several sources of uncertainty regarding these analyses. We focus mainly on three points: uncertainty due to user annotations, uncertainty due to the analysis method, and uncertainty due to the traceability of information from source level to binary level.

One concern are the assumptions that often have to be made about the environment in which the code runs. Analysis tools typically allow the user to specify properties that can affect the outcome of the analysis, like limitations on value ranges for inputs, or whether some variables should be considered volatile. There is always a risk that such manually-specified properties are false.

Another potential source of uncertainty is if the analysis is indeed unsound. A value- or program-flow analysis must always rely on some assumptions on the semantics of the code: if there are situations where these are not fulfilled, then value ranges or program flows may be underestimated which in turn can yield an unsafe WCET estimate. For instance it is not uncommon that analyses consider numbers to be unbounded, "mathematical" numbers when indeed they have a finite representation in the software. Fig. 1 shows an example where a loop bounds analysis that rests on this assumption will fail. Such an analysis will find that the loop body can be executed only one time, whereas in reality `i`, which is an 8-bit unsigned number, will wrap around from 254 to 0 when incremented by 2 causing a non-terminating loop.

Analyses that involve floating-point numbers can suffer from unsoundness since it may be hard for a tool to support all the varieties of floating-point arithmetics that different processors use. It is therefore common that analysis tools use some standard floating-point arithmetics such as IEEE floating-point arithmetics, or the native arithmetics of the computer where the tool executes. However, this may not be the arithmetics used by the target machine, which then can yield an unsound analysis.

```
unsigned char i;
i = 254;
while (i <= 255) do {
    i = i + 2;
}
```

Figure 1 A simple example of a code with wrap-around.

In a similar fashion, if low-level code is analysed, the analysis can become unsound if it assumes the wrong endianness of the target architecture.

A final potential source of unsound analysis are pointers. If the analysis cannot bound a pointer in a program point where the pointer yields the address for a write, then a sound analysis must assume (very pessimistically) that the write may occur anywhere in the memory (possibly including, for instance, the program code). Thus, after such a write, basically all information about what may happen next in the program is lost. It is therefore common that analyses assume that writes using such unbounded pointers cannot be more than "reasonably" out of bounds: for instance, it is commonly assumed that they cannot modify the program code.

For maximal confidence, value- and program flow analyses must be performed on the linked binary code. However, analyses can be hard to perform on this level due to lack of information about types, syntactic structure, etc. Therefore it is not uncommon that these analyses are attempted at the source code level instead, with the results subsequently being mapped to the binary level with the aid of debug information or alike. However, even the results of an analysis that is sound on the source level may not be sound for the compiled binary due to compiler optimisations changing the structure of the code. Even if optimisations are turned off, some compilers may still perform code transformations like turning `while-do` loops into `do-while` loops. Work has been done how to trace program flow constraints through compiler optimisations [16, 24], but production compilers do not implement these solutions.

2.3 Microarchitectural Timing Models

The low-level analysis step computes the worst-case execution times of code fragments, such as basic blocks. It is based on a cycle-accurate model of the target platform which specifies the hardware behavior when executing a sequence of instructions. There exist several ways to build such a model:

- The hardware timing model can be specified by the tool designer or by the end-user from the processor manual that is usually publicly available from the processor manufacturer. As mentioned in [27], this task is both time-consuming and error-prone due to: (a) missing or even incorrect documentation (user manuals generally focus on specifying the programming models but do not provide a detailed view of the processor internals nor accurate instruction timings), and (b) human errors when translating the natural-language description of the processor architecture given by the manual into a formal model. The reliability of such hand-crafted timing models is difficult to assess and this is even more true when the processor features complex hardware mechanisms, which are usually poorly documented.
- Measurement techniques can also be used to reverse engineer hardware parameters. In [14], monitoring registers are used while running specifically-designed micro-benchmarks to identify the processor's write and cache replacement policies. Similar techniques are used in [5] to investigate translation look-aside buffers (TLBs). New variants of the pseudo-LRU replacement policy implemented in the Intel Atom D525, the Intel Core 2 Duo E6750, and the Intel Core 2 Duo 8400 but not publicly documented could be discovered by application of automata learning [1] in case of of the Intel Atom and a combination of automatic measurements and human insight [2] in the other two cases. In [36], micro-benchmarking is used to discover the behavior of various components of an Nvidia GPU architecture, such as the warp scheduling policy. Note that all these approaches need manual work to (a) design micro-benchmarks that can exhibit hardware parameters, which might be particularly difficult in the presence of a totally original scheme that would not be described in the literature, and (b) interpret the results to determine how the processor or memory hierarchy works. In that sense, such techniques cannot provide fully reliable models but they can confirm, deny or complement the processor's description provided in the manual. In the first case, confidence in the model is increased.
- The timing model can be derived (semi-)automatically from a formal description of the hardware in a hardware description language, i.e., the microarchitecture's VHDL or Verilog model [28]. This hardware description contains the complete information required to build the microarchitectural timing model for timing analysis. Then, cumbersome and often error-prone reverse engineering is not required. Correctness of the timing model relative to the VHDL or Verilog model can be achieved and shown with comparably little effort. Due to the complexity of the hardware description and the various abstraction levels used to describe the microarchitecture, a completely automatic derivation is not possible. The timing model must be tight so as not to inflate the complexity of the microarchitectural analysis. The main obstacle remains the availability of the hardware description. Processor manufacturers are very reluctant to provide detailed hardware descriptions out of fear of plagiarism.

2.4 Microarchitectural Analysis

Given a microarchitectural timing model and a program, the task of microarchitectural analysis is to determine bounds on the execution times of program fragments. The main challenge for modern processors is that execution times of individual instructions strongly depend on the state of the microarchitecture. As an example, a memory instruction that causes a cache miss may easily take 100 times as long as one that causes a cache hit.

To correctly estimate the execution time of a program fragment, microarchitectural analysis thus needs to determine the set of states that the microarchitecture can be in when executing the program fragment. To do so microarchitectural analysis needs to take into account all possible program executions and initial states that may lead to a program fragment. To cope with the potentially very large number of cases, *abstraction* is employed where possible. Precise and efficient abstractions have been found for caches [8], whereas less structured components such as pipelines are mostly analyzed concretely, often leading to a very large number of states to be explored. Such analyses can be proven correct relative to a concrete model using the theory of Abstract Interpretation [7]. Such proofs have been carried out in paper and pencil proofs for caches and branch target buffers. Due to the lack of "strong" abstractions, beyond abstracting register and memory values, such proofs have been omitted for pipeline analyses.

To counter state-space explosion in microarchitectural analysis, it is tempting to only consider the local worst cases. Due to *timing anomalies* [20, 25], however, this is generally unsafe. It is an open problem to prove freedom of timing anomalies for models of realistic microarchitectures, while some success has been achieved in simplified scenarios [26].

Another approach to reduce analysis cost and possibly even improve precision is to analyze different components separately. For instance, one might attempt to analyze the pipeline separately from the cache. In the case of multicores a common approach is to separate the analysis of the bus blocking from the WCET analysis. Such approaches assume *timing compositionality* [11], i.e., that execution time can be safely decomposed into contributions from different components. As is the case with timing anomalies, some microarchitectures are conjectured to be timing-compositional [38], but none has formally been proven so.

A number of projects have focused on designs of or design principles for *timing-predictable* microarchitectures. This includes CompSOC [12], JOP [29], MERASA [33], Predator [38], and PRET [19]. Timing models for microarchitectures developed based on the principles identified in these projects are often simpler than those for commercial microarchitectures. This enables more precise and efficient analysis, and it also increases confidence in their correctness.

2.5 Path Analysis

Path analysis is the final step in WCET analysis. In this step, the results of microarchitectural analysis and ISA-level analysis are combined to reason about all possible program executions and their timing. Often, a program's control-flow graph (CFG) is used to bridge the gap between the two analysis levels:

- Microarchitectural analysis delivers bounds on the execution times of program fragments like the basic blocks of the CFG.
- ISA-level analysis delivers constraints on the possible paths that can be taken through the CFG, such as loop bounds.

The goal of path analysis is then to identify the worst-case execution path given the constraints obtained by ISA-level and microarchitectural analysis. Instead of explicitly exploring all paths, state-of-the-art WCET analyzers rely on an *Implicit Path Enumeration Technique* (IPET). Possible paths and their execution time bounds are expressed as the solutions of a set of integer linear constraints. The solution maximizing the execution time can then be found by an integer-linear programming (ILP) solver. Other path analysis approaches that have been considered are based upon SAT modulo theory or model checking.

The main source of doubt in path analysis comes from the fact that execution time is estimated in numbers of machine cycles. This is an integer value that is estimated by solvers using finite number representations. Some research verified the solution for LP and/or SMT solvers: the main idea is to verify the certificate corresponding to the optimum [9, 4]. As far as we know, these studies have not yet been extended to ILP: such a verification appears possible, but the problem to solve is larger.

2.6 Confidence in Tool Implementations

A potential source of uncertainty, which is common to more or less all analysis stages, is the possibility of bugs in the tool implementations. One way to reduce the uncertainty stemming from this is to make a formal verification of the algorithm, or the code of the implementation, using a proof assistant. By reducing the *trusted code base*, i.e., the part of the code that is not verified and thus has to be trusted, confidence can be gained.

One such effort has been done in the context of the CompCert certified compiler. Maroneze [22] developed a static WCET analysis tool, using previously known techniques, in the CompCert environment and provided a formal proof of correctness for those parts of this tool that correspond to ISA-level analysis and path analysis. Correctness proofs for the microarchitectural analysis were left as future work. This work demonstrates that this kind of formal verification is within reach also for complex WCET analysis tools.

3 On Confidence in Measurement-based WCET Analysis

Measuring a task's execution time is an alternative approach towards timing verification. It provides a simple and straightforward method to derive execution time estimates. Besides the simplicity of the measurement-based approach – which is in contrast to static timing analysis – no microarchitectural model is required. Measurements can be derived for the actual binary running directly on the target architecture, thus eliminating a prominent source of uncertainty. The very same hardware used in the embedded system can also be used for the measurement.

A fundamental drawback reduces the overall confidence in measurement-based approaches: It is practically infeasible to obtain measurements that cover the complete input space and the complete set of initial processor states, let alone interferences occurring during run-time. Exhaustive measurements are simply not possible for realistically-sized tasks and modern microarchitectures.

3.1 End-to-end Measurements

End-to-end measurements represent the most naive approach towards measurement-based timing verification. The execution time from task dispatch to completion is measured for a set of program inputs and initial processor states, and based on the highest measured execution times, WCET bounds are derived. These bounds are then multiplied by a safety margin to account for potential optimism in the measurements. This safety margin was the original motivation for mixed-criticality systems. A higher safety margin increases the WCET bound and so, also the confidence in it.

Path coverage techniques [39] are traditionally used to increase confidence in end-to-end measurements by automatically generating the set of test-cases. The automatically generated test vector is claimed to either cover all paths, or to cover at least the worst-case path. As these methods only treat the task input and not the initial processor states, doubt remains.

3.2 Hybrid Approach

Hybrid approaches [37, 17, 18, 31] use measurements to obtain estimates of the worst-case timing of program fragments. These are then combined during path analysis as in static WCET analysis tools to obtain an estimate of the WCET. There are two main approaches to obtain the timing of program fragments by measurements:

- By instrumenting the program code to obtain timestamps before and after executing each program fragment [37].
- By performing end-to-end measurements through different program paths, from which the execution time of each fragment can then be estimated [17, 18, 31].

Both approaches require the generation of inputs that drive execution through a given program point. Any unreachable code needs to be proven to be so, which may be difficult for deeply nested code.

The first approach may not deliver faithful execution-time estimates on pipelined processors, because timing is distorted through instrumentation. A too fine-grained instrumentation may also be impossible due to the large amount of trace data that must be captured at high pace. The second approach avoids these problems since it identifies the fine-grained timing models from end-to-end measurements. The approach in [18] works also for complex architectures, and can identify timing models with context-dependent costs for better precision. Both approaches may underestimate the WCET whenever execution times of program fragments depend on input data values or on the execution history, as the measurements will usually only cover strict subsets of the possible cases. On modern processors with deep pipelines, branch predictors, and caches this is the case. Exceptions are highly timing-predictable microarchitectures such as PRET machines [19].

While hybrid approaches are not guaranteed to be sound, and it is hard to quantify confidence in its results, they can also be pessimistic. The path analysis phase may combine observed worst-case timings of program fragments which may not occur together during a single program execution, which may be avoided in static analysis [32].

4 Beyond WCET Analysis – The Impact of Interference on Shared Resources

The WCET analysis exposed in previous sections considers a task that runs in isolation on the platform: its execution time is assumed not to be impacted by any external source. In practice, this assumption is rarely true: the task might be interrupted, or preempted by the scheduler for the benefit of a concurrent task, and the hardware state (e.g. cache contents) might be changed by the interrupt service routine or the preempting task; it may also be delayed by a hardware-level operation (e.g. a DMA transfer) or a task running on another core (in a multicore platform) that compete for shared resources (bus, memory, etc.). These last years, several approaches have been proposed to account for such interferences.

Techniques to estimate the cache-related preemption delay (CRPD), i.e. the number of additional cache misses due to context switching or periodic/sporadic interrupts, have received much attention recently [3, 6]. Most of the approaches use static code analysis techniques and suffer the same confidence issues as static WCET analysis: the model and/or its implementation might be flawed.

Multithreaded/multicore platforms raise additional issues: a task can experience increased latencies upon accesses to shared resources, due to conflicts with simultaneously running tasks; and the contents of shared storage resources, such as shared L2 caches, can be evicted by co-running tasks all along the task's execution, not only at preemption points. Evicted cache contents may result in additional delays to reload information that is still in use by the task. To estimate additional latencies due to interferences from other tasks, two strategies are possible: the *blind* approach assumes the worst possible co-running task set and considers absolute worst-case latencies [23, 15]; the *scheduling-aware* approach restricts the analysis of possible conflicts to the set of tasks that can effectively run together with the task under analysis [13, 21]. Both approaches require that the sharing control policy allow upper bounding delays; this is the case for a round-robin bus arbiter, for example. In the same way as for single-core architectures, the documentation of COTS multicores might not provide enough guarantees in the sharing scheme description to be fully confident in estimated delays. For this reason, designs for time-predictable multicores have been developed in recent projects, such as T-CREST [30] or parMERASA [34]. Note that a common assumption for most of the works on this topic is that the system features timing compositionality [11], which allows analysing each component separately. Unfortunately, this property is not easy to prove.

5 Discussion and Open questions

We have discussed possible sources of errors in different WCET analysis methods: static, measurement-based, and hybrid methods, and how the potential for such errors will affect the confidence in the result. The motivation comes from the need to quantify the confidence in WCET estimates for safety-critical systems, e.g., to select C(HI) and C(LO) in Vestal's model for scheduling of mixed-criticality systems.

All WCET analysis methods have potential sources of errors. For methods like static and probabilistic analysis, which rely on mathematical models, the risk that the models do not comply with reality must be taken into account. For methods that include measurements an additional source of uncertainty is the quality of the test vectors, and the ability of the method to find inputs provoking the longest execution traces with the highest instruction execution times.

Identifying sources of reduced confidence in WCET estimates is not difficult. *Quantifying* the confidence is much harder, and we do not attempt to make any detailed assessment of the different techniques in this regard. Having said that, we do believe that static analysis methods have an edge as regards the potential to obtain high confidence in that (1) given that the underlying models are correct, the methods are provably safe, and (2) since the methods are not based on measured data, confidence can be obtained by a thorough validation of the models against the real systems, and by verifying the correctness of the algorithms and tool implementations that build on the models. The latter can be done either by a formal verification, or by a run-time verification where checking of correctness certificates is integrated in the tools.

For methods that rely on measurements, confidence also rests on the quality of the test data. Better criteria are needed to assess this quality with respect to timing. Traditional coverage criteria, like path coverage, do not consider hardware effects on timing: we would need more refined coverage criteria that take hardware states, like cache contents, into account.

However, although hard, quantified confidence in WCET analysis methods is essential if models like Vestal's model are to be applied in the design of mixed-criticality systems. Our aim is to start a discussion on how this can be done.

Acknowledgements. This paper has been initiated in the Dagstuhl Seminar 15121 – Mixed Criticality on Multicore/Manycore Platforms.

References

1 Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

2 Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

3 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

4 Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.

5 Vlastimil Babka and Petr Tuma. Investigating cache parameters of x86 family processors. In *Computer Performance Evaluation and Benchmarking*. Springer, 2009.

6 Lee Kee Chong et al. Integrated timing analysis of application and operating systems code. In *34th Real-Time Systems Symposium (RTSS)*, pages 128–139. IEEE, 2013.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, Los Angeles, January 1977.

8 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

9 Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *20th International Symposium on Static Analysis (SAS)*, pages 345–365, 2013.

10 Patrick Graydon and Iain Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Workshop on Mixed Criticality Systems*, 2013.

11 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

12 Andreas Hansson et al. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM TODAES*, 14(1):1–24, 2009.

13 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium (RTSS)*, 2009.

14 Tobias John and Robert Baumgartl. Exact cache characterization by experimental parameter extraction. In *Int'l Conf. on Real-Time and Network Systems (RTNS)*, 2007.

15 Timon Kelter et al. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2), 2014.

16 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *22nd International Conference on Real-Time Networks and Systems (RTNS)*, page 97, 2014.

17 Markus Lindgren, Hans Hansson, and Henrik Thane. Using measurements to derive the worst-case execution time. In *Int'l Conf. on Real-Time Computing Systems and Applications (RCTSA)*, 2000.

18 Björn Lisper and Marcelo Santos. Model identification for WCET analysis. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

19 Isaac Liu et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*, September 2012.

20 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

21 Mingsong Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*, 2010.

22 André Oliveira Maroneze. *Certified Compilation and Worst-Case Execution Time Estimation.* PhD thesis, Université Rennes 1, June 2014.

23 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

24 Adrian Prantl. *High-level compiler support for timing analysis.* PhD thesis, Technical University of Vienna, June 2010.

25 Jan Reineke et al. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

26 Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in presence of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2009.

27 Marc Schlickling. *Timing Model Derivation – Static Analysis of Hardware Description Languages.* PhD thesis, Saarland University, January 2013.

28 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. In *Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2010.

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

30 Martin Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, To appear in 2015.

31 Sanjit A. Seshia and Alexander Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Trans. Embed. Comput. Syst.*, 11(S2):55:1–55:27, August 2012.

32 Ingmar J. Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs.* PhD thesis, Saarland University, May 2010.

33 Theo Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99, 2010.

34 Theo Ungerer et al. parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability. In *Euromicro Conference on Digital System Design (DSD)*, pages 363–370, 2013.

35 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

36 Petros Voudouris. Analysis and modeling of the timing bahavior of GPU architectures. Master's thesis, Eindhoven University of Technology, 2014.

37 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. 17:430–444, 2008.

38 Reinhard Wilhelm et al. 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.

39 Nicky Williams. WCET measurement using modified path testing. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2005.

Bare-Metal Execution of Hard Real-Time Tasks Within a General-Purpose Operating System

Georg Wassen[1] **and Stefan Lankes**[2]

1 **Operating Systems Research Group, Faculty of Electrical Engineering and Information Technology, RWTH Aachen University**
Aachen, Germany
`wassen@lfbs.rwth-aachen.de`

2 **Institute for Automation of Complex Power Systems, E.ON Energy Research Center, RWTH Aachen University**
Aachen, Germany
`slankes@eonerc.rwth-aachen.de`

Abstract

Integrating high performance and real-time demands on multi-processor systems is a challenging task. We present our concept of isolating processes from a general-purpose operating system without deeply invading modifications. This allows executing code on dedicated CPUs with minimum latency and jitter like bare-metal on micro-controllers. The unbounded execution of mixed critical processes on the same system induces performance interference in real-time tasks. We present the implementation of isolated partitions on multi-processor x86 systems running Linux and describe challenges restoring operating system stability. This work also presents our experience with Non-Uniform Memory Access architectures that allow to partition the system in a way that the impact to memory and I/O transfers of other partitions is minimized.

1998 ACM Subject Classification D.4.7 Real-time systems and embedded systems

Keywords and phrases hard real-time system, high-performance computing, non-uniform memory access, bare-metal execution

Digital Object Identifier 10.4230/OASIcs.WCET.2015.75

1 Introduction

Today, multi-processor systems are commonplace from High-Performance Computing (HPC) to small embedded systems. The shift from faster CPUs to multi-processors is more radical than any other architectural change of the last decades [22] because it requires the software paradigm to care for concurrency. This problem is split in two challenges: Firstly to synchronize correctly to avoid in order races and to get a correct result in every execution and secondly to avoid pitfalls that slow down the performance. HPC has a long head start in designing and optimizing concurrent software but this is mainly reserved to domain-specific experts [10, 11]. Evidently, multi-processor systems are also utilized for real-time applications and a large amount of research concentrates on improving the predictability of concurrent tasks in such systems. Various approaches exist that cover a range of applications from completely controlled to a mix of real-time and general-purpose loads. New implementation methods and tool kits are searched for to ease the development of efficient and race-free software. This is especially true for real-time systems, where the additional goal of predictability must be met.

Multi-processor systems generally include multiple CPUs that have access to the entire memory via a global address range. Multi-core processors are CPUs integrated in the same package, usually sharing the Last-Level Cache (LLC). If all processors have the same distance to the memory, the system is a Uniform Memory Access (UMA) architecture. In contrast, the Non-Uniform Memory Access (NUMA) architecture places multiple memory regions directly at groups of processors. These nodes have a lower latency and a higher throughput to their local memory but still can access the remote memory. This does not require special programming since all memory regions are laid out in a linear address range, but the performance can be greatly increased if the data is partitioned accordingly [5, 14].

Real-time systems generally require not only a correct result, but also have a timing requirement, usually a deadline after which the value of the result either vanishes abruptly or declines. We define *hard real-time* tasks to require absolutely no missed deadlines and *soft real-time* as more tolerant, e. g. to a certain percentage of missed deadlines. The former demand can only be proved by a formal verification while soft real-time tasks can usually be evaluated practically [17]. The design and verification is split into the path analysis of tasks, conditions and loops in a Control Flow Graph (CFG) and the run-time estimation of basic blocks [24]. The former includes the schedulability analysis of multiple tasks and must regard dependencies and communication. On multi-processor systems, the effort increases but solutions do exist. In contrast, the run-time estimation of small, linear parts of code – basic blocks – must regard architectural features that are often hardly documented or complexly interweaved on powerful commodity processors with multiple levels of caches. So far, this results either in the requirement of simplified models or in an overly pessimistic estimation.

1.1 Contributions

We consider homogeneous processors with a partitioning approach. This usually means either executing multiple OS instances in an Asymmetric Multi-Processing (AMP) layout or binding certain tasks to dedicated processors in a Bound Multi-Processing (BMP) concept. We present our implementation to provide hard real-time capabilities on a single instance of the general-purpose operating system Linux by completely isolating single processes on dedicated CPUs while the other processes are executed unaffected on the remaining CPUs. We extend our basic concept [23] and present detailed experience implementing benchmarks and demonstration applications. We share our experience with partitioning on NUMA systems. This architecture is well established in HPC but – in our opinion – under-represented in recent real-time research. The main contribution of this work is a demonstration of a NUMA system that allows to execute hard real-time tasks with very low jitter as well as compute-intensive applications on the same system. The architecture naturally minimizes the performance interference which simplifies the design and verification.

1.2 Structure

The next Section introduces to our partitioning concept, explains how we implemented fundamental services and presents our demonstration application. Section 3 explains how we use the NUMA architecture and presents the evaluation of memory access patterns. In Sec. 4, we transfer those findings to non-uniform I/O. Section 5 comments on related work and Sec. 6 concludes and provides and outlook to future work.

2 System Architecture

The term *hard real-time* is defined to guarantee a reaction below a defined maximum latency in every possible situation. A program or system can only be proved to comply to this constraint by a formal analysis of all possible code paths. Besides the code of the real-time tasks, there is other software competing for the processors, foremost the operating system's interrupts and system calls. The analysis of all possible code paths in the OS is beyond practicability for a current general-purpose operating system.

Therefore, we isolate a single hard real-time process on each dedicated CPU. By avoiding system services, we implemented a bare-metal execution that relieves the time-sensitive code from external influences. The usability is restored by providing user-mode communication and device access that allows to interact between isolated tasks and usual processes without introducing new timing dependencies. The remaining influence of hardware effects will be analyzed in the following Sections.

2.1 Application

The isolation concept was implemented by adhering to implementation rules, configuring the base system properly and finally deactivating all interrupts on the isolated CPU. The following system-wide configuration settings are reasonable to support all real-time applications. These settings are available on the x86 architecture, but similar effects are present on other hardware and can usually be addressed by similar means. In our evaluation, some settings can be done in the firmware configuration (BIOS or UEFI setup), others are selected at the Linux boot command line or configured as runtime settings.

Symmetric Multi-Threading (Intel HyperThreading) is deactivated because the logical processors share some execution resources and the local cache in an unpredictable way [6]. The System Management Mode is a special feature of x86 processors to enable the firmware to execute routines for power management, security, etc. It executes at the highest privilege level and can not be influenced or deactivated by the operating system [9]. Experiments on various systems have revealed a duration of System Management Interrupts between 5 μs and 250 ms on all CPUs concurrently. Their use depends on the firmware that may use them frequently. Some can be configured, e. g. by deactivating the legacy keyboard support but every system must be assessed for its acceptability in this regard. The CPU frequency can be adapted by the operating system or the processor itself. These features must be configured to remain at a constant rate. Linux provides an NMI Watchdog that sends non-maskable interrupts to non-responsive CPUs. This feature must be deactivated. Like every real-time and embedded system, all services of the operating system should be evaluated and those not required should be deactivated.

To isolate hard real-time tasks on their dedicated CPUs and guarantee the servicing of all remaining processes, a partitioning concept based on CPU Affinity or the CPU-Set feature was used. CPU-Sets are a convenient mechanism to create groups of CPUs and assign processes to them. That way, isolated partitions can be created for each task. An application partition allows a BMP layout for soft real-time execution of other, less time-critical or best-effort tasks and a system partition gathers all remaining processes and services. To ensure the execution of interrupts, they are assigned either to the CPU of the system partition or to several dedicated CPUs by the means of IRQ Affinity.

The real-time process must be implemented according to the usual recommendations using preallocated buffers and memory locking to avoid page faults and the proper distribution of jobs and the implementation of algorithms suited for real-time. Of course, the application must be correct and error free. The isolated real-time tasks finally clear the interrupt flag to avoid all interruptions. Further, they must avoid all system calls during the real-time operation because those are of unpredictable timing in a General-Purpose Operating System (GPOS).

2.2 Services

For the described approach of isolated tasks on bare-metal processors, the applicability is restored by providing user-mode communication and device access. To interact with the physical world, many real-time applications require reading sensor data and controlling actors. These can be directly connected to I/O adapters that convert analog voltage to digital values and vice-versa. Operating system drivers are not allowed to be used by isolated tasks because their latency would be too large and too unpredictable. But as adapters are controlled either by I/O instructions or by memory-mapping their configuration registers, these devices can also be operated directly by user-mode processes. Devices for real-time applications usually provide libraries for low latency user-mode access.

The initialization and allocation of resources can remain under the control of the operating system and should be done before the time sensitive phase of the program begins. During run-time, the reading and writing of values must then be done directly. Given that interrupts are deactivated, the programming can not follow the event-based paradigm but must use polling. Real-time capable libraries provided by hardware vendors must be assessed for not using system calls during run-time. Network drivers could also be realized in user-space. Shared buffers must be mapped at initialization time and the event-based processing of incoming data must be converted to a polling realization. Further, the network stack must be made real-time capable and running in user-space.

We used shared memory for lightweight communication between threads and processes. During the initialization phase, page-table entries directing to the same physical page frames are created. This hardware based method does not require any operating system support after set-up. The caches are coherent by hardware design. As in all concurrent applications, shared resources must be protected from simultaneous access to avoid using invalid data. If a shared variable (flag) is written always by the same task, no synchronization is required. For complex data structures and message queues, the synchronization could be realized with atomic operations. But hard real-time tasks could be blocked by mutexes held in lower prioritized tasks (priority inversion). This can be solved by applying wait-free algorithms [12].

To simplify application development, we implemented a communication and synchronization library based on shared memory. During the start of a process, the shared memory segment is set up using System V shared memory. This is part of the UNIX specification and available on POSIX compliant operating systems. Within that shared segment, the needed synchronization primitives such as flags and message queues are allocated. The shared resources are protected by atomic operations for mutual exclusion and wait-free algorithms. The design of wait-free message queues is possible [19] and provides a Worst-Case Execution Time (WCET) that is limited independently from other users of the object. This restores the communication capabilities of the isolated process under hard real-time terms.

2.3 Operating System Modification

The deactivation of all interrupts on some CPUs leads to the instability of the underlying Linux system. Among the observed problems are other CPUs locking up due to synchronous inter-processor callbacks, unreliable system wall clock time, and increasing kernel memory consumption. We modified the Linux kernel 3.9 imitating the CPU hotplugging to deactivate all subsystems on a CPU to isolate a task. With this modification, the task does not need to clear the interrupt flag to ensure uninterrupted operation. On x86 systems, clearing the interrupt flag can only be done by the code running on the CPU itself, while our mechanism can be triggered externally. This allows to reactivate a non-responsive isolated CPU.

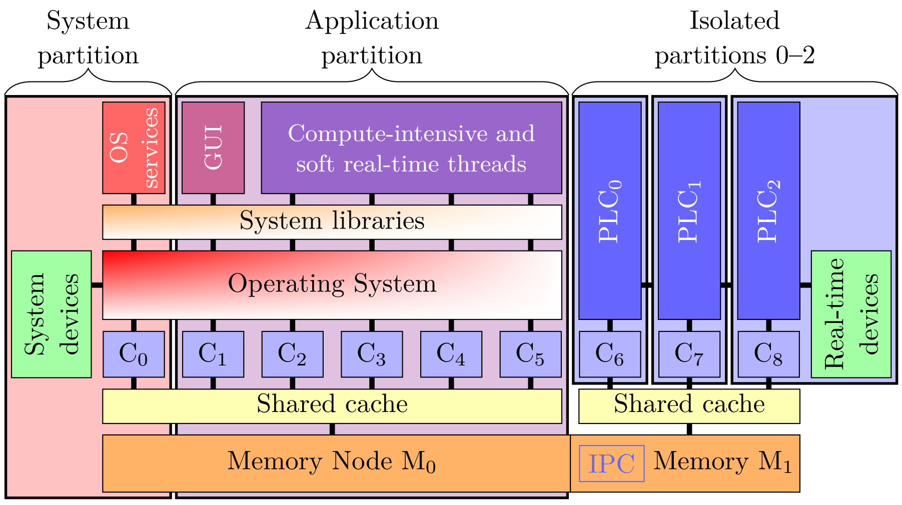


Figure 1 Example application architecture with bare-metal tasks.

2.4 Example

An example application partitioning is shown in Fig. 1. The GPOS with its system services is executed in a dedicated partition. The application partition is used for the main part of the application to execute Graphical User Interface (GUI), compute-intensive tasks and data management. These two partitions can also be combined or split into multiple dedicated partitions depending on the requirements. It is important to create a single hard real-time partition or even dedicated isolated partitions for each CPU where hard real-time tasks are isolated from the compute-intensive loads.

The system and application partitions execute arbitrary services based on standard libraries and use the OS drivers to access devices that have lower timing requirements or that are throughput bound. The isolated partitions use shared memory and wait-free algorithms [12] to communicate with the other partitions and to access their peripheral devices directly using on user-mode drivers or direct I/O. This ensures the lowest latency and the least impact of the GPOS and the remaining application onto the isolated tasks. The isolated tasks can be executed bare-metal or in an embedded Real-Time Operating System (RTOS) implementing user-mode threads.

We implemented this concept on a twelve processor machine with two NUMA nodes. The System and Application partitions are assigned to the processors of the first NUMA node and three isolated partitions are set up on the second node. The remaining CPUs on that node are left idle. This application was developed with an industry partner and is in successfully used in production. The benchmarks described in the following Sections are implemented using the same architecture.

3 Non-Uniform Memory Access

All processors in a Uniform Memory Access (UMA) system share the system bus and the Memory Controller (MC). Cores sharing the inclusive LLC of a multi-core package may even cause the eviction of cache lines from other core's local caches [13]. In contrast, a Non-Uniform Memory Access (NUMA) system has multiple memory partitions attached directly to MCs located in the multi-core packages. These packages are connected by Interconnect

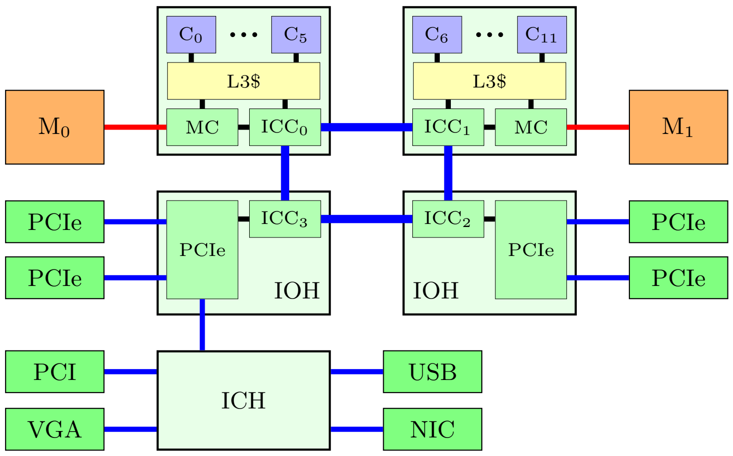


Figure 2 NUMA system architecture of Tyan S7025 main board with two CPU sockets and two I/O bridges.

Controllers (ICCs) that forward memory requests for remote addresses. With the practical knowledge, that shared functional units are potential contention points, we designed the partitioning of the real-time application respecting the NUMA nodes. The system and (soft real-time) applications are executed on different nodes than the isolated tasks.

Our example system (Fig. 2) has two processor sockets each equipped with a six core Intel Xeon E5645 processor. The example application introduced in Fig. 1 maps the system services to core C_0 and the main application to the remaining cores C_1 to C_5 of the first NUMA node. The hard real-time tasks are executed in isolation on dedicated cores of the other NUMA node (C_6 to C_{11}). By default, new memory allocations are placed locally so that the system and the application access mainly the memory node M_0 while the memory regions of the isolated partitions are placed on the other node. Only a small Inter-Process Communication (IPC) buffer is accessed across the interconnect link between ICC_0 and ICC_1.

The benchmark application records the loop execution time for accessing a memory buffer of varying size. At the same time, the main application generates different loads on the other partition. The isolated tasks communicate their state and current timing via wait-free message queues to the main application and watch for a termination signal like in a real application. Figure 3 shows the jitter experienced by the isolated task. The result is independent from the load range as displayed exemplarily for 16 KiB (L1$), 8 MiB (LLC) and 512 MiB as long as the isolated task has a memory usage well below the cache size. Only the memory usage of the isolated task itself increases its own maximum latency if it uses a large part of the LLC. The load does not cause any performance interference on this NUMA system.

The increasing jitter caused by using large parts of the LLC matches the experience on UMA systems. Since every NUMA node is a multi-core processor, their cores impact each other in the same way as all processors in a UMA system do. Therefore, the isolated tasks should restrict themselves as much as possible to the private Second-Level Cache (L2$) of 256 KiB. In this case their maximum latency remains reliably low. Even occasional cache misses compulsorily caused by shared IPC buffers do not interfere fatally. The worst latency of main memory accesses is in the order of 10^6 CPU cycles (Millisecond range) but occurs very seldom and only if the isolation writes very large memory buffers. Minimum and average

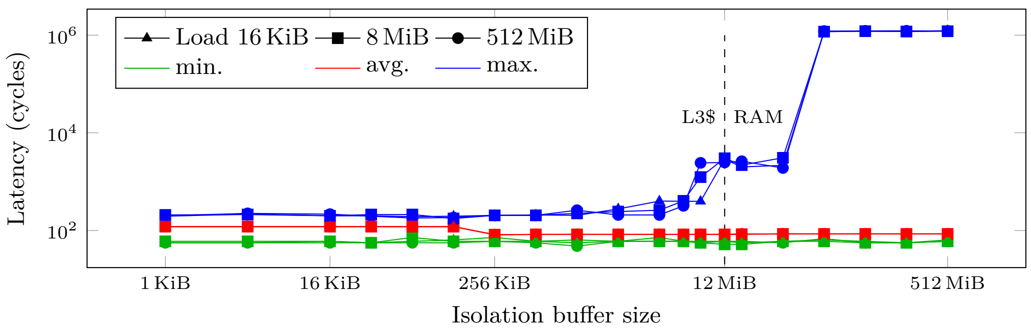


Figure 3 Jitter caused by load on other NUMA node.

latency are not impacted by those events. Our experiments have shown, that the number of events in the range of the maximum latency increases linearly with the isolation buffer size.

4 Non-Uniform I/O

Many hard real-time systems with high frequency and low latency requirements are programmable logic controllers that require direct interaction with physical signals. This can be accomplished on the x86 architecture with data acquisition expansion cards. Their access time in the order of several thousand CPU cycles (Microsecond range) is low compared to what the processor could calculate in this period. But similarly to memory access, the jitter caused by device access is even worse on multi-processor systems because of system bus contention [21]. To solve the problem of unpredictable jitter, co-scheduling of tasks [16] and hardware extensions [1] have been proposed. Since we aim for unrestricted concurrent execution on available systems, these approaches are inapplicable.

Analogous to the memory transfers, in UMA systems all I/O traffic uses the same system bus. In the same way that NUMA architectures are capable of separating contending memory transfers of system applications and time-critical tasks, they also offer a solution for predictable I/O transfers. Additionally, PCI-Express (PCIe) offers prioritized data streams that can be transferred concurrently over switched point-to-point connections without interfering [2]. Like the NUMA architecture's natural partitioning capabilities improving the timing predictability of memory accesses, we expect that the PCIe connections in NUMA architectures are better suited for hard real-time systems than PCI and UMA.

The main board architecture shown in Fig. 2 has two I/O bridges (IOH) that are connected to the CPU sockets in a QuickPath interconnect forming a ring. We therefore connect the peripheral devices used by the real-time tasks to the PCIe slots connected to the right-side IOH. The system partition controls the standard devices (graphics adapter, USB, Ethernet) that are provided by the ICH which is connected to the left IOH. The only data transfers crossing the partition border are still the memory transfers to the IPC buffers.

The test series uses memory-mapped I/O accesses to different devices that are connected to the built-in PCI slot managed by the ICH and to a PCIe slot of the real-time partition. We also measured a PCIe to PCI bridge installed in the real-time partition and one of the PCIe slots assigned to the system partition but those gave similar results as the real-time partition PCIe connection. The benchmark was executed in isolated tasks and recorded a statistic of latencies during high memory and I/O loads executed in the system partition.

The results are displayed in Fig. 4. The large I/O overhead of the x86 architecture in the range of 2000 to 5000 CPU cycles (1–2 μs) covers outliers and jitter of I/O transfers.

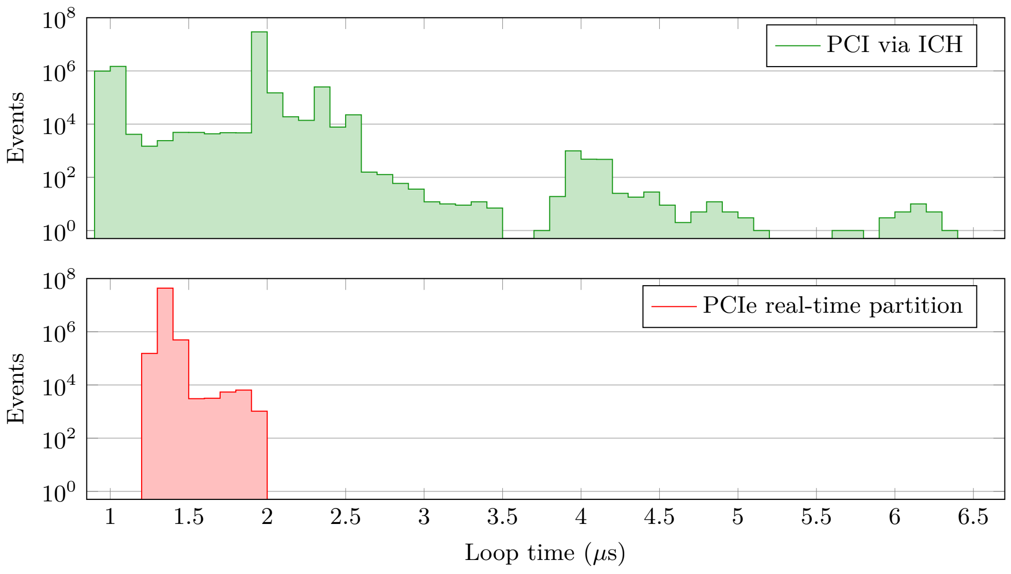


Figure 4 Distribution of access latencies to different I/O locations under system and I/O load.

The PCI device connected to the chipset's ICH has access latencies up to 6 μs under high system load while the access to the PCIe devices is more predictable between 1.2 and 2 μs. The results of the other PCIe partition and a PCIe to PCI bridge in the real-time partition are very close to the lower histogram with a constant offset of approx. 0.2 μs. The PCIe devices are not influenced by the load. The transfers using the PCIe interconnect can be completed in bounded time while the ICH (former south bridge) is easily contended. It remains as future work to verify the interconnect architectures and to evaluate more systems and more recent CPU architectures. However, we have completed extensive test series with uninterrupted execution times up to 72 hours that support these results on our test system.

5 Related Work

Real-time systems with multiple processors have been researched thoroughly [4, 18]. Current approaches [25] to tightly estimate the execution time on multi-processor systems either simplify the architecture [3] or restrict the execution on the entire system by strictly controlling all tasks [15]. In contrast, we aim for allowing arbitrary applications in the system partition and present a method to restrict performance interference. A very deep analysis of the causes of jitter in x86 systems is presented by Dasari et. al. including the memory bus and caches [8] as well as I/O [7]. Some publications mention NUMA systems in their plans for future work [7, Sect. IV.C] or just explain theoretical foundations [20] but they do not analyze the opportunities of such systems. All evaluations of multi-processor I/O so far only regard UMA systems and the older PCI bus. Stohr includes PCIe in his analysis of using the x86 architecture for real-time systems [20] but does not cover NUMA and non-uniform I/O. Despite an extensive literature review, these works cover only UMA systems and even if mentioning NUMA systems, those are only briefly touched and either dismissed or left for future work. To the best of our knowledge, no publications present experience with hard real-time applications on current NUMA systems do exist so far.

6 Conclusion

We demonstrated the value of the NUMA architecture that is underrepresented in hard real-time systems research, so far. With the spacial partitioning of AMP and BMP real-time systems to distinct processors, NUMA systems allow to also partition both memory and I/O transfers. Extensive test series fortified the assumptions derived from HPC experience and architectural analyzes. However, this is only shown for the used CPU architecture (Intel Westmere) and main board (Tyan S7025). Nevertheless, it indicates that a careful design which considers the NUMA architecture is in fact capable to reduce the performance interference and to provide hard real-time guarantees on commodity x86 systems without restricting the load in the system partition.

This concept was implemented in a Hardware-in-the-Loop simulator for industrial integration testing. Compute-intensive high-frequency hard real-time tasks could successfully be ported from special micro-controllers to a standard x86 server system. Other application fields are FPGA codes, that could be integrated easier with existing complex applications to create mixed critical systems.

Acknowledgments. This research was supported by the Chair for Operating Systems at RWTH Aachen University.

References

1 Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-Time Control of I/O COTS Peripherals for Embedded Systems. In *Proc. 30th IEEE Real-Time Systems Symp. (RTSS)*, pages 193–203. IEEE, 2009.

2 Ajay V. Bhatt. Creating a PCI Express Interconnect. White paper, Technology and Research Labs, Intel Corp., Santa Clara, CA, USA, 2002.

3 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Proc. 25th Int. Conf. Architecture of Computing Systems (ARCS)*, volume 7179 of *LNCS*, pages 98–110. Springer, 2012.

4 Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*, pages 157–169. IEEE, 2008.

5 Timothy Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 1–18, 1993.

6 Steve Brosky and Steve Rotolo. Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux. In *Proc. 2003 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE, 2003.

7 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the Sources of Unpredictability in COTS-based Multicore Systems. In *Proc. 8th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 39–48. IEEE, 2013.

8 Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, et al. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *IEEE 10th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068–1075. IEEE, 2011.

9 Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM Rootkits: A New Breed of OS Independent Malware. In *Proc. of the 4th Int. Conf. on Security and Privacy in Communication Networks (SecureComm)*, pages 11:1–11:12. ACM, 2008.

10 Agner Fog. *Software optimization manuals.* Technical University of Denmark, Copenhagen, Denmark, 2013. `http://www.agner.org/optimize/`.

11 Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Inc., Boca Raton, FL, USA, 2010.

12 Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

13 Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches. In *Proc. 43nd Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, pages 151–162. IEEE, 2010.

14 Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proc. Int. Symp. on Memory Management (ISMM)*, pages 11–20. ACM, 2011.

15 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, et al. A Predictable Execution Model for COTS-based Embedded Systems. In *Proc. of 17th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 269–279. IEEE, 2011.

16 Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*, pages 221–231. IEEE, 2008.

17 Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In Raimund Kirner, editor, *8th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 8, Dagstuhl, Germany, 2008. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

18 Simon Schliecker and Rolf Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, 2011.

19 Philippe. Stellwag, Alexander Ditter, and Wolfgang Schröder-Preikschat. A Wait-Free Queue for Multiple Enqueuers and Multiple Dequeuers Using Local Preferences and Pragmatic Extensions. In *Proc. 4th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 237–248. IEEE, 2009.

20 Jürgen Stohr. *Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme.* Doctoral dissertation, Fakultät für Elektrotechnik und Informationstechnik, TU München, Munich, Germany, March 2006.

21 Jürgen Stohr, Alexander von Bulow, and Georg Färber. Bounding Worst-Case Access Times in Modern Multiprocessor Systems. In *Proc. 17th Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 189–198. IEEE, 2005.

22 Herb Sutter. The Free Lunch Is Over. *Dr. Dobb's Journal*, 30(3), March 2005.

23 Georg Wassen, Stefan Lankes, and Thomas Bemmerl. Harte Echtzeit für Anwendungsprozesse in Standard-Betriebssystemen auf Mehrkernprozessoren. In Wolfgang A. Halang, editor, *Herausforderungen durch Echtzeitbetrieb*, Informatik aktuell, pages 39–48, Berlin, Germany, 2012. Springer.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, et al. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

25 Reinhard Wilhelm and Jan Reineke. Embedded Systems: Many Cores – Many Problems. In *Proc. 7th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 176–180. IEEE, 2012.

Analysing Switch-Case Code with Abstract Execution*

Niklas Holsti[1], Jan Gustafsson[2], Linus Källberg[2], and Björn Lisper[2]

1 **Tidorum Ltd.**
Helsinki, Finland
`niklas.holsti@tidorum.fi`

2 **School of Innovation Design and Engineering, Mälardalen University**
Västerås, Sweden
`{jan.gustafsson,linus.kallberg,bjorn.lisper}@mdh.se`

Abstract

Constructing the control-flow graph (CFG) of machine code is made difficult by dynamic transfers of control (DTC), where the address of the next instruction is computed at run-time. Switch-case statements make compilers generate a large variety of machine-code forms with DTC. Two analysis approaches are commonly used: pattern-matching methods identify predefined instruction patterns to extract the target addresses, while analytical methods try to compute the set of target addresses using a general value-analysis. We tested the abstract execution method of the SWEET tool as a value analysis for switch-case code. SWEET is here used as a plugin to the Bound-T tool: thus our work can also be seen as an experiment in modular tool design, where a general value-analysis tool is used to aid the CFG construction in a WCET analysis tool. We find that the abstract-execution analysis works at least as well as the switch-case analyses in Bound-T itself, which are mostly based on pattern-matching. However, there are still some weaknesses: the abstract domains available in SWEET are not well suited to representing sets of DTC target addresses, which are small but sparse and irregular. Also, in some cases the abstract-execution analysis fails because the used domain is not relational, that is, does not model arithmetic relationships between the values of different variables. Future work will be directed towards the design of abstract domains eliminating these weaknesses.

1998 ACM Subject Classification F3.2 Semantics of Programming Languages: Program Analysis

Keywords and phrases dynamic control flow, indexed branch, machine-code analysis, WCET analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.85

1 Introduction

Static analysis of the worst-case execution time (WCET) of a program usually begins by building the control-flow graphs (CFG) of all subprograms, and the call-graph connecting the subprograms. On the machine-code level, where most conventional WCET tools work, the tool has to find the possible successor instructions of each instruction under analysis. This is easy when the instruction defines its successors statically but hard for control-transfer instructions with dynamic target addresses, for example register-indirect jumps. Such *dynamic transfer of*

* This work was supported by the APARTS project in the FP7 Marie Curie IAPP programme (grant agreement no. 251413), and by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

control (DTC) instructions often result from switch-case statements. Other reasons for DTC include function pointers, virtual function calls in object-oriented languages, and closures in functional languages.

This paper focuses on DTC from switch-case statements. This analysis is typically local and intra-procedural, while function-pointer analysis is a global value analysis and assumes that the CFGs of all procedures are already known. The switch-case statement in languages such as C or Ada is a very flexible control structure. The programmer can choose the type of the switch index, for example an 8-bit or a 32-bit number; whether the cases are numbered densely 1 .. n or are a sparse subset of a large range; whether each case is reached by a unique index value or by a set or range of values; and whether there is a default case or not. Compilers often generate quite different kinds of code to implement different kinds of switch-case statements. Moreover, hand-written assembly code (still often found in run-time libraries) can have quite tricky switch-like structures. Switch-case statements are also common in real programs and a failure to analyse even one of them can prevent the analysis of the whole program. Of course, the compiler knows all the targets of each switch-case DTC, and in principle could make this information available to a WCET analyzer, but in practice we cannot expect to get this information from all the industrial compilers, and especially not for manually coded assembler subprograms – even debugging information is sometimes unavailable for those.

On the surface, analysing a DTC instruction is just value-analysis: we need to know the possible run-time values of the variable or register that holds the final target address. However, in practice we have more requirements – the analysis of the DTC generated from a switch-case statement should:

- produce the precise value-set, because any over-estimation will introduce false execution paths, which can make a mess of the rest of the analysis,
- produce the sequence of instructions that leads to each case, so that later steps in the analysis can find an accurate WCET for each case,
- connect each case with the corresponding values of the switch index, again for use in later analysis steps (for example to find bounds for a loop that is nested in a case and depends on the switch index),
- apply uniformly to most types of code for most kinds of switch-case statements, and be robust to changes in the code as the compilers evolve or manually written code is revised. Adding new patterns or other special cases for every new version of every compiler for every supported processor is cumbersome and expensive.

The more general analysis-based methods have a larger chance of satisfying the last point, while more specialized pattern-based methods will suit the first three points better.

How can a value-analysis method meet these requirements, in particular a method using abstract interpretation? Firstly, for high precision a very suitable and accurate abstract domain must be used, and imprecision induced by merging abstract values must be limited. Secondly, because switch-case code is often optimized and quirky, the "smooth" and theoretically appealing abstract domains such as intervals or polyhedra may not work well. Thirdly, this code often uses memory-resident tables, so the analysis must model memory values as well as registers. However, these tables usually contain constants, so a model of indexed access to constant memory data may be enough, and mutable memory data need not be modelled.

Here, we apply the *abstract-execution* method of the SWEET tool to switch-case code, and compare it to the switch-case analyses in the Bound-T tool, which are in part pattern-based. This is also an experiment in modular tool design where a general value analysis tool is used as a plugin to a WCET analysis tool. Indeed, this scheme should also be applicable to other value- and WCET analysis tools if they are provided with suitable interfaces.

SWEET requires ALF code [5] as input, not machine code. Thus, we needed a front-end to translate machine-code into ALF. We implemented this function in Bound-T, which in the future will let users of Bound-T benefit also from other SWEET analysis results like loop bounds and other flow-facts. However, so far we have implemented the link between Bound-T and SWEET only for DTC analysis.

The rest of this paper is organized as follows. Section 2 introduces our tools and methods: abstract execution, SWEET, Bound-T, and the coupling of Bound-T and SWEET to use abstract execution for DTC analysis. Section 3 describes the test programs we used, in particular the machine-code form of their switch-case statements. Section 4 discusses the results of the analysis of the test programs, tracing the reasons for successes and failures. Section 5 briefly surveys some related work on analysing DTC. Section 6 draws some conclusions for future development of SWEET and Bound-T.

2 Tools and methods

2.1 SWEET and Abstract Execution

SWEET [10] was originally aimed at WCET analysis. The current version of SWEET focuses on value-analysis and control-flow analysis. SWEET analyses programs represented in the ALF language [5]. ALF is a textual language, similar to a compiler's intermediate representation, and is designed to represent code on both low- and high level (like machine code, or C) faithfully. An ALF program consists of global data declarations and functions. A function consists of local data declarations and a sequence of assignment statements interleaved with branch statements, where any statement can be labelled for use as branch target.

In addition to the conventional value analyses based on abstract interpretation with widening, SWEET provides *Abstract Execution* (AE) [6]. This can be seen as a very context-sensitive value analysis without widening, where different loop iterations are analysed separately. This makes the analysis resemble a symbolic execution where the program is executed with abstract states, in the abstract domain, rather than in the concrete domain. This yields a more precise value-analysis, but has the draw-back that it may not terminate.

During the AE, several abstract states may appear. When the AE reaches a control-flow join, it may or may not *merge* the abstract states from the incoming paths using the least upper bound operator in the abstract domain. The merging is controlled by several command-line options and can even be completely shut off. Disabling merging has the effect that the abstract states that provide the result of the value analysis are not merged, which effectively turns the abstract domain into its powerset domain. This makes it possible to obtain sparse value sets from the AE even when a regular abstract domain such as intervals is used. This matters because DTC target addresses often form sparse sets.

The current version of SWEET supports both intervals and *Circular Linear Progressions* (CLP) [13] as abstract domains. CLP combines the well-known intervals with *congruences* [4]: the latter can represent non-unit address strides, which is important for DTC analysis.

Our aim is to use AE, with no merging, to compute the possible values of the target addresses of the DTC instructions in the machine-language program under analysis. The question then arises how DTC instructions can be represented in an ALF program.

2.2 Dynamic control flow in ALF

ALF statements can have labels which have a symbolic part and a numeric offset part. Such *statement references* are first-class ALF values and can be computed dynamically by selecting a value for the symbolic part from those existing in the program, and/or by numerically computing a value for the offset part. The computed statement reference can then be used in a jump statement – an ALF DTC. SWEET's analysis can produce a safe (i.e. possibly over-estimated) set of possible DTC targets, as ALF statement references.

However, when the ALF code is generated from machine code, modelling a DTC with ALF statement references would require the ALF code to include all the ALF statements representing the instructions that may be the targets of the DTC. Therefore these instructions would have to be found and translated into ALF, before the ALF representation of the DTC could be subjected to analysis. In effect, this would require the ALF generator itself to resolve the DTC, before generating the ALF program.

The reader may ask, why not translate *all* of the machine code, as found in the code-memory image, into ALF statements? Then all possible DTC targets would be present in the ALF program. However, the presence of data intermixed with the code, and the existence of variable-length instructions and the consequent ambiguity of where instructions start, and the possible presence of more than one instruction set or encoding in the same program mean that this approach could generate a very large ALF program, much of which would be nonsense because it would not represent real instructions. Furthermore, an ALF program is syntactically divided into functions (subprograms), and an ALF DTC jump is permitted only within an ALF function, not across functions. A translation of the whole program into ALF would have to group the instructions into functions, which is impossible without resolving the DTCs in the machine code, or without some meta-information on function boundaries in the machine code. Similar reasons prompted Theiling's use of bottom-up CFG reconstruction [14] as opposed to the top-down, decode-all-the-image approch used by some earlier work.

Therefore, we allow ALF programs to be incomplete. We make the ALF program end at the point of an unresolved DTC, and analyse the incomplete program to discover the values of the numerical variables on which the DTC target depends. The analysis results are then used to compute the DTC target addresses. These addresses are passed back to Bound-T, which in turn generates a new ALF program that includes code reachable from the resolved DTC's. This iteration goes on until all DTC's are resolved – see Section 2.4.

2.3 Bound-T and its analyses of dynamic control flow

Bound-T is a WCET and stack-usage analyser using pure static analysis of linked, executable machine-code programs [11]. Bound-T's DTC analyses are always implemented or initiated by processor-specific and sometimes compiler-specific analysis procedures, triggered by specific code patterns. Still, the specific DTC analyses build on and use the general analyses in Bound-T. The DTC analyses fall into two groups:

- *Abstraction-based analyses.* Here, the target-specific part of the analysis models the DTC in a form that is amenable to one of the abstraction-based analyses in Bound-T: constant propagation, value-origin (*def–use*) analysis, and Presburger-arithmetic analysis. The latter is incidentally described in [8] and is based on the Omega Calculator [16].
- *Partial-evaluation analyses* [7]. Here, the target-specific part of the analysis is responsible for starting, stopping and parametrizing the partial evaluation.

The main limitation of the abstraction-based analyses is that the Presburger-arithmetic analysis cannot model dynamically addressed memories, only statically identified variables.

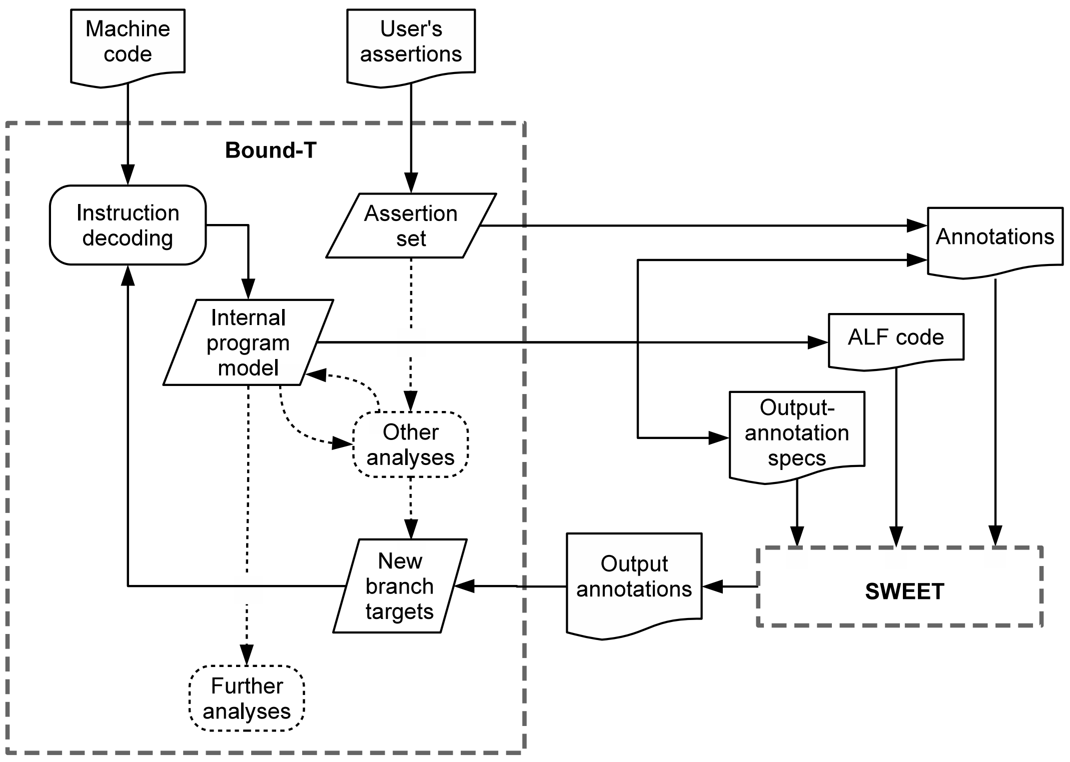


Figure 1 Overall analysis flow.

If the DTC involves a table of some form, the Presburger-arithmetic analysis must be focused on the addresses of the table elements, while the reading and interpretation of the data in the table must be done separely and depends on the DTC type.

The main limitation of the partial-evaluation analyses is the need to detect where and how the evaluation should be started – typically on entry to specific library functions (“switch handlers”). Some manual reverse engineering of these functions is needed, to understand how they access the “switch-tables” encoding the switch-case statements.

2.4 Coupling SWEET and Bound-T

To evaluate abstract execution for DTC analysis, we extended Bound-T to export its internal program model into ALF and to run SWEET on that ALF program, asking for AE using either the interval domain (which is sensitive to variable bit-widths and wrap-arounds) or the CLP domain, no merging, and output of the abstract values of the variables which determine the targets of the DTC’s. (However, when merging is disabled, the values produced are sets of concrete values rather than abstract values.) Bound-T then reads the SWEET output, computes the corresponding DTC target addresses, and adds those instructions and their static successors to its internal program model, if necessary fetching and decoding more code from the executable under analysis. The extended program model may contain new DTC’s. This process (ALF generation — SWEET analysis — completion of program model) is iterated until it stabilizes. Fig. 1 illustrates the data flow in this process. The iterative construction and extension of program models involving DTC was already implemented in Bound-T for its own DTC analyses, so essentially we only added SWEET as a further method of DTC analysis. For technical details see [9].

We implemented this coupling for the Atmel AVR 8-bit processor architecture, a widely used microcontroller family [1]. The AVR is a difficult subject for DTC analysis because it implements mostly 8-bit operations and no general $base + index$ addressing. The AVR has a Harvard architecture with separate instructions for reading data from code memory. Code addresses are 16 or 22 bits wide (depending on the AVR device) and are manipulated as pairs or triples of 8-bit data. Some of the 32 general 8-bit registers can be paired and used as 16-bit registers, but the set of 16-bit operations is quite limited. The AVR machine code generated for switch-case DTC is complex and varied. This makes pattern-matching analyses unreliable and favours semantically based analyses, such as AE.

3 Test programs

We selected 6 programs from the Bound-T test suite and added one new program to the suite, giving a total of 7 test programs (P1 – P7 below). The programs can be provided on request without IP restrictions. The goal was to include many different forms of switch-case code using DTC. Note that it is the machine-code form of the DTC that is important, not the source-code form. We did not use the well-known MRTC benchmarks because their switch-case statements produce simple or no DTC code with the compilers we have tried. The switch-case structures in our test programs are either extracted from real programs, or written to be similar to switch-case statements in real programs. The DTC machine codes are representative of Tidorum's experience with real programs and several different compilers. The chosen programs are small, because the prototype implementation of ALF export in Bound-T cannot yet handle complex parameter-passing confidently. However, analysis of switch-case code is for the most part local and intra-procedural so the program size is not important.

P1 A bottom-test loop, which contains a dense switch-case statement implemented by an indexed dynamic jump into a table which, in turn, contains static jumps to the code for each case. The loop counter is 8 bits, running from 15 to 19, and the cases are also numbered 15 to 19 with no default case.

P2 A dense switch-case statement with an 8-bit index, implemented by loading the 16-bit jump target address from a table in code memory, in two 8-bit parts, and executing an indirect jump to that address. The cases are numbered 0 to 9 plus a default case.

P3 A sparse switch-case statement with an 8-bit index, four cases plus a default, implemented by a switch-table and the corresponding switch-handler subprogram. This is a simplified, artificial switch-table structure from the example in [7].

P4 A sparse switch-case with an 8-bit index, four cases plus a default, implemented with the real switch-table structure (sparse variant) and real switch handler used in the IAR Systems C compiler.

P5 A dense switch-case statement with an 8-bit index, ten cases plus a default, implemented with the real switch-table structure (dense variant) and real switch handler used in the IAR Systems C compiler.

P6 A 16-case switch, implemented by an indexed jump into a dense table of jumps, in which the 4-bit index is assembled from two 2-bit pieces using "rotate" followed by "or". In effect, this is a 2-dimensional switch-case with two indices of 2 bits each. Jump-table elements are two addressing units long.

P7 A variant of P6 in which the "rotate" instruction is replaced by a "left shift" instruction and jump-table elements are one addressing unit long.

Table 1 Analysis results.

Program	*Bound-T result*	*SWEET (interval) result*	*SWEET (CLP) result*
P1	*Exact result*, but see note.	*Exact result.*	*Exact result.*
P2	*Fails.*	*Fails.*	*Fails.*
P3	*Fails.*	*Exact result.*	*Exact result.*
P4	*Exact result.*	*Exact result*, but see note.	*Exact result*, but see note.
P5	*Exact result.*	*Fails.*	*Fails.*
P6	*Fails.*	*Fails.*	*Exact result.*
P7	*Exact result.*	*Exact result.*	*Exact result.*

We know of some forms of switch-case code that are not represented in this test set: tables of *code offsets*, instead of code addresses, and address (or offset) tables accessed by *hashing* the case index, instead of using the case index itself (minus some lower bound) as an offset into the table. There are probably more strange forms lurking in the code jungle.

4 Analysis results

We compiled the test programs into AVR executables (choosing either the gcc or the IAR compiler, to generate the desired kind of DTC machine code) and analysed them with Bound-T alone (disabling the use of SWEET) and with the Bound-T + SWEET combination (disabling Bound-T's own DTC analyses) using either the interval domain or the CLP domain, with no merging over paths. For these small programs, no analysis lasts over 10 seconds on a MacBook Air with a 2.13 GHz Intel Core 2 Duo processor. Table 1 shows the results. For a detailed discussion of each success and failure, refer to [9].

For P1, Bound-T alone needs a manual assertion that the switch-case index is non-negative. The SWEET domains have better models of signedness and wrap-around.

For P2, Bound-T alone fails because the arithmetic analysis cannot model general addressable memories, and because a specific pattern for this "load-address-from-table" idiom is not implemented. The interval domain fails because lack of congruence information leads to over-estimation of the octet pointer into the address table, which causes such huge over-estimation of the DTC targets that Bound-T rejects the solution. The CLP domain gives the exact set of strided octet pointers into the address table, but the set of address values cannot be exactly merged into a single CLP value (such merging of table values cannot be disabled in SWEET). This causes significant over-estimation of the DTC targets, which in the third iteration leads to such an over-estimated CFG that SWEET's AE fails to terminate in a reasonable time, probably because a spurious eternal loop was introduced.

For P3, Bound-T alone fails because this switch handler is an artificial one for which Bound-T has no detector. Otherwise the partial evaluation method would work here.

For P4, SWEET with either domain gives the exact result for the DTC, but the WCET is overestimated. Bound-T's partial-evaluation method fully unrolls the search loop in the switch-handler, letting Bound-T find the exact worst-case path. The analysis with SWEET retains the loop and Bound-T then calculates the WCET under the normal assumption that all loop iterations use the worst-case path through the loop body. This WCET overestimation could be avoided either by transporting more flow-facts from SWEET to Bound-T, or by letting SWEET compute the WCET as a program variable [8].

For P5, the interval domain fails because it is not relational and does not include congruence. The CLP domain fails because it, too, is not relational, and because it does not see repeated additions of the same value (for example $x + x$) as equivalent to multiplication ($2x$) and therefore loses congruence information.

For P6, Bound-T alone fails because the carry-out from the “rotate” operation is not well modelled in the arithmetic analysis. SWEET with the interval domain fails because lack of congruence leads to over-estimation of the pointer into the jump table, leading to over-estimation of the DTC targets. However, in this contrived and simple case, the over-estimation only doubles the set of targets, so Bound-T accepts the result, leading to an apparently successful analysis but an over-estimated CFG and WCET. The CLP domain succeeds because the jump instructions in the jump table lie at regularly spaced addresses, which can be represented exactly in a single CLP value.

For P7, the exact result from Bound-T alone depends on some unsound assumptions in modelling left-shifts. SWEET succeeds with the interval domain because the unit stride of the jump-table elements makes congruence analysis unnecessary.

In total, Bound-T alone succeeds on 4 and fails on 3 programs. SWEET’s analyses succeed on 5 and fail on 2 programs. If the analyses are combined, only P2 is left as a failure. Further patterns and special cases could easily be added to Bound-T to correct the failures, including P2, but that would just be one more step on the endless trail of special cases.

The results show that the abstract-execution method is very promising, working in essence as well as and in some cases better than the set of processor- and compiler-specific and manually constructed pattern-based methods in Bound-T. Comparing the two numerical abstract domains, the CLP domain succeeds in one case (P6) where the interval domain fails. We expected that CLP would yield a larger improvement, but its benefits are often masked by the direct translation from machine instruction semantics into ALF code, which hides congruence information when, for example, a C source instruction such as `y=2*x` is translated into two AVR instructions that do `y=x; y=y+x`. The Presburger-arithmetic analysis in Bound-T, being relational, is able to combine the two instructions and conclude that $y = 2x$. When Sen and Srikant introduced the CLP domain [13], they combined it with the domain of affine equalities, perhaps to counter this sort of problem.

Some of Bound-T’s failures are due to poor modelling of signedness and wrap-arounds. The Presburger-arithmetic formalism considers all variables to be signed, unbounded integers. Wrap-around can be modelled with conditional (disjunctive) formulae, but if that is done systematically the time and space requirements explode. In contrast, SWEET’s interval and CLP domains handle wrap-around gracefully (but not always exactly). However, because we disable merges in SWEET, it may also run into scalability problems for large programs.

The failure of SWEET on P5 is interesting. P5 computes the address of the table element for the DTC target address in three steps. Step 1 computes an intermediate value from the switch index. Step 2 compares the switch index to the range of case numbers. If the switch index is in range, step 3 computes the final address using the switch index and the intermediate value. Because the intermediate value is computed before the switch index is known to be in range, and because the domains are not relational, SWEET can put no useful bounds on the intermediate value, and so the final address is also unbounded.

5 Related work

Our work follows the “bottom-up” CFG reconstruction approach as also described by Theiling [14]. However, it seems that Theiling does not expect strong value-analysis to be used for switch-case code, saying that the “decoders may use pattern matching and code slicing to detect switch tables”. Cifuentes and Van Emmerik [3] use slicing and symbolic expression substitution to simplify switch-case code so that it can be matched to a library of "normal forms", an advanced and flexible form of code-pattern matching. However, their experiments use wide-word, powerful processors of the SPARC and Pentium class, avoiding the complications of 8-bit microcontrollers such as the AVR.

The CodeSurfer/x86 tool described by Balakrishnan and Reps [2] has many similarities with SWEET and some with Bound-T. CodeSurfer/x86 analyses Intel-x86 machine code. The initial control-flow and call-graphs are generated by the commercial IDAPro disassembler [12], which may leave some switch-case DTCs unresolved. A value-analysis based on abstract interpretation, using a domain of intervals with strides (congruence), provides possible DTC targets for an iterative extension of the CFG, as in our method. However, CodeSurfer/x86 is at present limited to the Intel-x86 instruction set and some of the decisions made during the analysis seem specific to this architecture. The value-analysis uses only 32-bit quantities, and it is not clear how well it models unsigned computation. The basic model is signed.

Balakrishnan and Reps [2] agree with two of our conclusions: that a domain modelling non-unit strides is necessary for this analysis, and that non-relational domains can cause important loss of precision. However, as we have seen, quite many DTCs have sets of target addresses which cannot be represented exactly as strided intervals. When merging is disabled, AE can produce more precise results.

6 Summary and conclusions

Focusing on the analysis of switch-case code using DTC, we compared SWEET's AE (with no merging) to the set of special, pattern-based analyses in Bound-T, on seven programs expressed in Atmel AVR machine code. AE won, on average, but failed for some programs because the numerical domain we used (CLP) is not relational. Other failures are due to the inability of intervals, even with congruence, to model sparse integer sets precisely.

Tidorum Ltd aims to make the SWEET-based DTC analysis a standard feature of Bound-T. This will also let Bound-T use other SWEET results such as flow facts. The main challenge is to make Bound-T's data-memory model, and its translation to ALF, sound with respect to aliasing. We also found weaknesses in Bound-T's handling of signedness and wrap-around. Tidorum aims to improve this, but the problem is complex if the model should be relational (as the current Presburger-arithmetic model is).

This work was partly responsible for prompting the implementation of the CLP domain in SWEET. Future work on SWEET, to better support the analysis of DTC's, includes the design of light-weight relational domains that can capture enough of the dependencies between variable values to gain the necessary precision. It also seems interesting for this purpose to have an abstract domain that can represent sparse sets of integers up to some predefined size of the set.

A more general question is whether it is possible and desirable to perform abstract interpretation with different domains, for different variables or different parts of the program. The DTC problem requires high precision, but usually deals with small sets of values, so the optimal domain for DTC analysis is different from that for problems which can do with less precision but cover huge ranges or sets of values. For maximal precision, one possibility is to convert abstract states into sets of concrete states, compute with transfer functions on these sets formed by applying the concrete transfer functions elementwise, and converting back to the original abstract domain when suitable. If the sets of concrete states do not grow too big, then this is tractable. Cofibered domains [15] provide a theoretical framework for this kind of use of multiple abstract domains.

References

1 The Atmel AVR. `http://en.wikipedia.org/wiki/Atmel_AVR`.

2 Gogul Balakrishnan and Thomas Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

3 C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Proc. Seventh International Workshop on Program Comprehension*, pages 192–199, 1999.

4 Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 3:165–199, 1989.

5 Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In Niklas Holsti, editor, *WCET'09*, pages 1–11, Dublin, Ireland, June 2009. OCG.

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

7 Niklas Holsti. Analysing switch-case tables by partial evaluation. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, 2007.

8 Niklas Holsti. Computing time as a program variable: a way around infeasible paths. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, Prague, Czech Republic, July 2008.

9 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Combining Bound-T and SWEET to analyse dynamic control flow in machine-code programs. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, November 2014.

10 Björn Lisper. SWEET – a tool for WCET flow analysis (extended abstract). In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 6th International Symposium on Leveraging Applications of Formal Methods (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485, Corfu, Crete, October 2014. Springer-Verlag.

11 Tidorum Ltd. Bound-T time and stack analyser. `http://www.bound-t.com`.

12 Hex-Rays SA. IDA disassembler. `https://www.hex-rays.com/products/ida`.

13 Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proc. 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07)*, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

14 Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7th International Conference on RealTime Computing Systems and Applications (RTCSA'00)*, pages 23–30, Cheju Island, South Korea, 2000. IEEE.

15 Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In Radhia Cousot and David A. Schmidt, editors, *Proc. Third International Symposium on Static Analysis (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer Berlin Heidelberg, Aachen, Germany, 1996.

16 W. Pugh et al. The Omega Project: Frameworks and algorithms for the analysis and transformation of scientific programs. `http://www.cs.umd.edu/projects/omega`.

Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs*

Jordy Ruiz and Hugues Cassé

IRIT – Université de Toulouse, France
`{jruiz, casse}@irit.fr`

Abstract

Worst-Case Execution Time (WCET) is a key component to check temporal constraints of real-time systems. WCET by static analysis provides a safe upper bound. While hardware modelling is now efficient, loss of precision stems mainly in the inclusion of infeasible execution paths in the WCET calculation. This paper proposes a new method to detect such paths based on static analysis of machine code and the feasibility test of conditions using Satisfiability Modulo Theory (SMT) solvers. The experimentation shows promising results although the expected precision was slightly lowered due to clamping operations needed to cope with complexity explosion. An important point is that the implementation has been performed in the OTAWA framework and is independent of any instruction set thanks to its semantic instructions.

1998 ACM Subject Classification J.7 Real time

Keywords and phrases WCET, infeasible paths, SMT, machine code

Digital Object Identifier 10.4230/OASIcs.WCET.2015.95

1 Introduction

Temporal properties verification is utterly important for critical embedded systems like avionics, automotive or any system involving real-time constraints. Such verification is usually performed in two steps: first, the Worst-Case Execution Time (WCET) of each task composing the system is computed and, second, this WCET is used to check if the set of tasks is schedulable according to the system real-time constraints. Hence, the WCET determination must be (a) safe (higher than the real worst-case execution) to ensure soundness of the scheduling and (b) tight or precise to minimize the hardware required to run the system.

This paper addresses the computation of WCET by static analysis that ensures safety of WCET by construction at the price of tightness: an overestimation of the WCET is produced without guaranty about precision. A successful and widely used approach to compute statically WCET is Implicit Execution Path Technique or IPET. It models the execution paths of the program and the host hardware (that induces execution time) as an Integer Linear System, ILP, whose maximization function is the estimated WCET. Lots of work has been devoted to model the hardware and, for some classes of microprocessor parts (LRU caches, pipeline, etc.), provides precise results.

Yet, modelling the execution paths seems to remain a challenge and could be an important source of precision improvement. Basically, IPET models the execution paths by a Control Flow Graph, CFG, where vertices represent blocks of instructions[1] and edges, execution flow between blocks. The CFG is usually derivated from a program by a basic analysis examining

* This work was partially supported by the french ANR project, W-SEPT.
[1] Usually machine instructions to be as close as possible to the hardware.

the branch targets of control instructions. Therefore, the set of paths represented by the CFG (potentially unbounded if the program contains loops) includes (a) all actual paths of the program and (b) other paths that are infeasible because of the program internal logic.

Applying the IPET approach to the CFG model of execution paths is consistent (as long as a maximum bound is provided for loops) because it consists in computing the maximum of execution paths. Adding infeasible paths to the set of actual execution paths will, in the worst case, produce an estimated WCET higher than the actual WCET. Yet, the added execution paths may cause a loss of tightness (distance between real WCET and estimated WCET) that is difficult to evaluate. In practice, separating feasible and infeasible paths faces the complexity issue of the number of paths in the program and in the CFG.

This paper proposes a new approach to detect infeasible paths in machine language programs based on (a) the representation of program states as labelled sets of predicates and (b) the detection of unsatisfiable conditions using a Satisfiability Modulo Theories (SMT) solver. The result is a mostly minimal list of infeasible paths or families of paths that may be used in WCET computation like IPET. The experimentation shows that our approach on the Mälardalen benchmark [4] is promising. It does not allow capturing all infeasible paths of the program but it should help improving precision of the estimated WCET.

The next section presents works related to infeasible path detection or execution path validation. Section 3 gives an overview of the proposed method and of its context while section 4 enters into details of the proposed analysis. Section 5 exposes the experimentation results and we conclude and propose future development in the last section.

2 Related Works

Several works already exist on this topic and are surveyed here.

Suhendra et al. [6] present an integrated approach to compute the WCET while excluding infeasible paths. The program is expressed as a Direct Acyclic Graph (DAG) and traversed in a bottom-up way. Throughout the analysis, the worst-case execution time is computed by accumulating block time and selecting the maximum time. During this traversal, predicates are built reflecting assignments and conditions, and paths with unsatisfiable set of predicates are discarded. This approach uses a very limited set of predicates that reduces analysis time but also the amount of infeasible paths found. In addition, there is no plan to cope with hardware effects.

Henry et al. [5] propose another integrated solution that represents as an SMT the semantic of execution paths and the time consumed by a particular path. To determine the WCET, one has to propose an upper bound, to add the matching predicate and to check the feasibility of the system. Main drawbacks of this approach are the computation time, the need for unrolling loops and the lack of support for non-scalar memory structures like arrays. In addition, there is no clear solution to cope with hardware effects except adding constant access time to instruction blocks. Unlike our approach, this approach determines the set of feasible paths and uses it to compute WCET.

The PathCrawler [7] tool generates disjunctions of conjunctions for any control point of the CFG, then proceeds to do constraint solving in order to rigorously verify properties. However, it runs on source code, and while we use a similar approach, our final goal differs: the detection of infeasible paths. In the next section, we present our approach that (a) is applied to machine code and (b) tries to preserve precision of predicates.

3 Proposed Method

3.1 Source of infeasible paths

3.1.1 Infeasible path patterns

We have identified three common types of infeasible paths found by our analysis. This list is not exhaustive and we sometimes find more complex infeasible paths but these are rarer. The first type of infeasible path simply comes from two mutually exclusive conditions in the source code, tested in sequential order:

```
if(x == 1)
        /*...*/
if(x == 2)
```

This can look like a mistake from the programmer, but there may be a lot of code between the two conditions, making the infeasible path much less obvious to see, and/or difficult to remove.

Some other infeasible paths are due to the restriction of the function parameters domain at a particular call and require interprocedural analysis to be identified. The `if(x)` condition will never be taken in the case of a call from `main()` in the following example:

```
void icrc(int x) {
        if(x)
                /*...*/
}
int main() {
        icrc(0);
```

The last type of common infeasible path is due to short-circuit condition evaluation during the compilation:

```
if(x && a)
        /* ... */
if(x && b)
```

In this case, the binary program may include an infeasible path because `if(x && a)` will be broken down into `if(x) { if(a)` by the compiler, unless smart optimizations are performed.

3.1.2 Analysis on Machine Language

Working on machine code allows precisely understanding the work of the program in the hardware and improving the WCET tightness. Yet, the analysis of the machine code suffers from (a) the lack of expressivity of machine instructions, (b) the size of the program and (c) the implementation complexity of data flow analyses. For example, registers are loosely typed and the structure of data in memory is not explicit.

Even though finding infeasible paths is easier on programs in high-level languages, an additional pass is required to map and exploit this information on binary programs. This pass is even more difficult if the optimizations of the compiler are activated. Even worse, analysis of the high-level language requires different analysers for each language, the complete sources to be available and prevents the processing of low-level subprograms written in assembly.

Hence, we choose to work at the machine code level but, as the real-time embedded domain uses lots of different microprocessors, we have to cope with as many instruction sets. Fortunately, the OTAWA C++ framework provides an abstraction of the different instruction sets through an independent *semantic language* [3]. It consists in the translation of the machine instruction into a sequence of semantic instructions that try to capture as precisely as possible its semantics. When the instruction cannot be expressed, a special `SCRATCH` instruction allows marking the result register as modified in an unspecified way. This approach has been successfully applied to translate several instruction sets like PowerPC, ARM, Sparc and Tricore.

Basically, the *semantic language* looks like a RISC-instruction set and is made of:

- three-operand computation instructions: `ADD`, `SUB`, `CMP`, `SHL`, `SHR`, etc;
- dedicated instructions to load a literal in a register: `SETI`;
- memory access instructions: `LOAD`, `STORE`;
- branches or assignments to program counter: `BRANCH`;
- conditional execution of the sequence: `IF`.

Most instructions work only on registers but temporary registers may also be used to let instructions exchange information without modifying the actual program state. In the following, we show how the *semantic language* can be used to perform infeasible path analysis.

3.2 Analysis overview

The lookup of infeasible paths is performed by top to bottom traversal of the program; all non-cyclic paths are explored, each machine instruction is broken down into one or several semantic instructions, and an abstract representation of the program state is updated accordingly for each path. Sets of possible program states are represented as lists of predicates, initially empty ($\top$), meaning that any state is possible.

3.2.1 Predicate generation

Figure 1 shows several examples of analysis results. (a) is an example of predicate generation, upon parsing an ARM instruction `mov r4, r0`. The first bold line is the ARM instruction, semantic instructions are capitalized, and italic lines are modifications to the list of predicates. Two predicates are generated (noted with $\oplus$), and t_1 is a temporary register introduced by OTAWA. While it may look purposeless here, OTAWA automatically chooses to use this intermediate to handle more complex uses of the `mov` instructions such as, for example, `mov r0, r1, LSR #2`. Here, the instruction `mov r4, r0` is translated into two semantic instructions, `SET t1, r0` and `SET r4, t1`. In turn, their interpretation generates two predicates, `t1 = r0` and `r4 = t1`.

3.2.2 Predicate update

The predicates can be modified either because the instruction we are parsing requires us to or because we wish to and believe it will be beneficial.

Extending the example (a), there are cases where modifying predicates is unnecessary but useful (b) because of the ephemerality of the temporary register t_1, whereas sometimes updating the predicate is required by the instruction (c). In (b), `[r4 replaces t1]` means that every occurrence of the temporary register t_1 will be replaced by register r_4. The predicate `r4 = t1` becomes identity (`r4 = r4`), so we only keep `r4 = r0` as a result of this `mov`.

In the example (c) initially containing "$r_3 = [SP - 12]$", in order for our predicate list to remain a valid abstract representation of the program state, this instruction that adds 1 to r_3 requires replacing every r_3 occurrence with $r_3 - 1$, hence the `[r3 - 1 replaces r3]` line.

```
mov r4, r0
  SET t1, r0
    ⊕ t1 = r0
  SET r4, t1
    ⊕ r4 = t1
```

(a) An addition.

```
mov r4, r0
  SET t1, r0
    ⊕ t1 = r0
  SET r4, t1
    ⊕ r4 = t1
  [r4 replaces t1]
  ⊖ t1 = r0
  ⊕ r4 = r0
```

(b) A useful update.

```
{ r3 = [SP-12] }
add r3, r3, #1
  SETI t1, 1
    ⊕ t1 = 1
  ADD r3, r3, t1
    [r3 - 1 replaces r3]
    ⊖ r3 = [SP-12]
    ⊕ r3 - 1 = [SP-12]
```

(c) A necessary update.

```
[...]
SET r13, t3
  ⊖ r13 = SP - 4
  ⊕ r13 = SP + 0
  ⊖ tempvars t1, t2, t3
```

(d) A removal.

Figure 1 Example of operations on the predicates.

3.2.3 Predicate removal

Instructions may force us to remove predicates from our list. For instance, if we set to r_{13} a new value, we must remove any predicate mentioning r_{13} as it will no longer be valid. In the example (d), the previously generated `r13 = SP - 4` predicate must be removed (noted with ⊖). In this case, the analysis identifies t_3 to the constant $SP + 0$ (value of the stack pointer at the start of the program), so it chooses to generate `r13 = SP + 0` in place of `r13 = t3`.

Also, temporary registers are always deleted at the end of every semantic instruction block (matching an assembly instruction), thus we should remove all predicates containing them, hence the ⊖ `tempvars t1, t2, t3` in the example.

4 Analysis Definition

4.1 Abstract domain

The goal of this analysis is to build a function $\mathbb{L} \to \mathbb{M}^\#$ that gives, at any control point $\mathbb{L}$, an abstract state $\mathbb{M}^\# : \vee_i(\wedge_j(\phi_{i,j}(R, M_h, M_s)))$ where:

- $R \cong \mathbb{Z}^n$ is the set of registers, n being the amount of available registers on the architecture;
- M_h represents the heap memory, $M_h \cong \mathbb{Z}^\mathbb{Z}$;
- M_s represents the stack memory, $M_s \cong \mathbb{Z}^\mathbb{Z}$.
- $\phi_{i,j}$ are predicates defined as:

$$\begin{array}{ll}
\phi : & Operand \times Opr \times Operand \\
Opr : & = \mid \neq \mid \leq \mid < \\
Operand : & Operand\ \omega\ Operand \mid -\,Operand \\
\omega : & + \mid - \mid \times \mid / \mid \mathrm{mod}
\end{array}$$

Each conjunction of predicates represents a set of possible program states for one or several execution paths up to a given control point. A control point may contain information coming from several paths: we do not merge the conjunctions of predicates but instead keep them in a set, this way $\mathbb{M}^\#$ is actually a disjunction of conjunction of predicates, and overestimates the set of possible program states.

In order to show this, we define $\mathbb{M}$ as a concrete program state, that is, the set of values of the registers and of the memory, and the following concretisation function $\gamma : \mathbb{M}^\# \to 2^\mathbb{M}$:

$$\gamma(\bigvee_i \bigwedge_j \phi_{i,j}(R, M_h, M_s)) \coloneqq \bigcup_i \bigcap_j \gamma'(\phi_{i,j}(R, M_h, M_s))$$

where for any p : Predicate, $\gamma'(p) \coloneqq \{m \in \mathbb{M} \mid p(m)\}$ and the analysis is sound if

$$\gamma(f^\#(s)) \sqsupseteq f(\gamma(s))$$

with $f : \mathbb{M} \to \mathbb{M}$ being any semantic instruction and $f^\# : \mathbb{M}^\# \to \mathbb{M}^\#$, also later noted $U(f)$, the corresponding modifications to be applied on our abstract state.

4.2 Update function

Our update function takes a semantic instruction and an abstract program state as parameters to return a new abstract state:

$$U : I_\mathbb{M} \times \mathbb{M}^\# \to \mathbb{M}^\#$$

where $I_\mathbb{M} = \mathbb{M} \to \mathbb{M}$ is the set of semantic instructions, which modify the machine state. Let Var be the set of variables, including R, M_h, M_s and the temporary registers $Temp$.

In the first place, we define the following functions to operate on $\mathbb{M}^\#$:

$$\texttt{invalidate} : \mathit{Var} \times \mathbb{M}^\# \to \mathbb{M}^\#$$

This function removes any *Predicate* containing a reference to the provided *Var*. Before removing, it computes the transitive closure of the current predicates to avoid unnecessary loss of information. Importantly, we observe for any $s : \mathbb{M}^\#$,

$$\gamma(s \setminus \{p\}) \sqsupseteq \gamma(s)$$

that is, by removing a predicate we lose precision but the result remains sound. It is a convenient property to handle unsupported operations. We also use variable replacement mechanics on predicates: for any x : *Var*, arithmetic expression e and program state $s : \mathbb{M}^\#$, the program state in which every occurrence of x has been replaced by e is noted $s\ [e\ /\ x]$.

For any instruction modifying a variable d, if the computation of the new value of d is independent of its previous value, we remove all predicates containing d as they become obsolete. If, at the contrary, this computation depends on the previous value of d, we try to apply to the affected predicates the inverse operation, but this inverse only exists if the original operation is bijective. For instance, $f : d \mapsto 3d$ is not surjective in $\mathbb{Z}$, therefore f^{-1} does not exist and we will not be able to update our predicates. In this case, we often have to throw away information about the variable f is applied to, although we can sometimes keep some properties: for example f preserves the parity in this case.

For any variables d, a, constant k:

$$U\,[\texttt{SETI d, k}]\ s \coloneqq \text{“}d = k\text{”} \cup (\texttt{invalidate}\ d\ s)$$
$$U\,[\texttt{SET d, a}]_{d=a}\ s \coloneqq s$$
$$U\,[\texttt{SET d, a}]_{d\neq a}\ s \coloneqq \text{“}d = a\text{”} \cup (\texttt{invalidate}\ d\ s)$$

For any variables d, a, b such that $d \neq a$, $d \neq b$:

$$U\,[\texttt{ADD d, a, b}]\ s \coloneqq \text{“}d = a + b\text{”} \cup (\texttt{invalidate}\ d\ s)$$
$$U\,[\texttt{ADD d, a, d}]\ s \coloneqq s\ [d - a\ /\ d]$$
$$U\,[\texttt{ADD d, d, b}]\ s \coloneqq U\ [\texttt{ADD d, b, d}]\ s$$
$$U\,[\texttt{ADD d, d, d}]\ s \coloneqq \text{“}d\ \%\ 2 = 0\text{”} \cup (s\ [d/2\ /\ d])$$

In the `ADD d, d, d` case, we lose information when replacing d with $d/2$, thus we add a predicate that says that d is even, to strip away the ambiguity.

$U\,[\texttt{MUL d, a, b}]\ s \coloneqq \text{“}d = a * b\text{”} \cup (\texttt{invalidate}\ d\ s)$
$U\,[\texttt{MUL d, a, d}]\ s \coloneqq \text{“}d\ \%\ a = 0\text{”} \cup (s\ [d/a\ /\ d])$
$U\,[\texttt{MUL d, d, b}]\ s \coloneqq U\ [\texttt{MUL d, b, d}]\ s$
$U\,[\texttt{MUL d, d, d}]\ s \coloneqq \text{“}0 \leq d\text{”} \cup (\texttt{invalidate}\ d\ s)$

Again, we add a predicate to avoid the loss of information in the `MUL d, a, d` case. We cannot express the square root, so there is not a whole lot to be done for `MUL d, d, d`. There is no support for the binary instructions `AND`, `OR`, `XOR`, thus for any d, a', b':

$U\,[\texttt{[AND/OR/XOR] d, a', b'}]\ s \coloneqq U\,[\texttt{SCRATCH d}]\ s \coloneqq \texttt{invalidate}\ d\ s$

This list is incomplete and there are many other semantic instructions to be interpreted (namely logical arithmetic shifts, memory accesses...), but these are handled in a similar way.

4.3 Flow analysis

Our analysis is applied on a **Control Flow Graph** (CFG), a directed rooted graph $G = (V, E, \epsilon)$ composed of lists of sequential instructions grouped in basic blocks which are the vertices (V) of said graph, and edges ($E = V \times V$) between basic blocks that represent the possible execution paths of the program. These edges can be conditional or not, and the CFG always includes one entry ($\epsilon \in V$) as a virtual basic block.

We parse the CFG with a working list algorithm, which general idea is to only process basic blocks once all the sources of the incoming edges have been processed. Below is a first version of the flow analysis algorithm for loopless programs:

```
wl <- {ϵ};
While wl != ∅
        pop bb from wl;
        If allIncomingEdgesAreAnnotated(bb)
                For edge in bb.ins
                        sl <- sl ∪ edge.getAnnotation();
                For state in sl
                        state.processBasicBlock(bb);
                For edge in bb.outs
                        new_sl <- ∅;
                        For state in sl
                                new_state <- state.processEdge(edge);
                                new_sl <- new_sl ∪ new_state;
                        edge.annotate(new_sl);
                        wl <- wl ∪ edge.target;
        End If
End While
```

We first initialize our working list `wl` to the root, then pop basic block elements from it until it is empty. For each basic block the algorithm is asked to work on, all its incoming edges are checked for *annotations* that give an abstraction of the program state at that control point. If any edge is missing an annotation, we cannot process the basic block yet and pospone it. Otherwise, we put all the annotations into a list of abstract states named `sl` above. Then, we update the states of `sl` to represent possible program states at the end of the basic block. Lastly, we annotate all the outgoing edges with this state list, update them accordingly if it is a conditional edge and add the block the edge points to the working list if it is not already in it.

The improved version of this algorithm that supports loops uses one more type of annotation: on loop headers, we remember the last abstract program state we have computed. We also keep information in our enhanced program states on whether we have reached a fixpoint in the most inner loop or not. If not, we follow all the edges except for the loop exit ones. If we have found a fixpoint, we follow all the edges except for back edges.

We merge the abstract program state list into one state on every loop header. When the state list becomes too big, we may also choose to merge all paths into one to save analysis time and prevent combinatorial explosion: this is a tradeoff between execution time and efficiency of the algorithm. Our merge algorithm is a trivial intersection with a few simple enhancements. For any conjunctions of predicates $p := \bigwedge_i(\phi_i)$ and $p' := \bigwedge_j({\phi'}_j)$ our merge operator $\sqcap : \mathbb{M}^\# \to \mathbb{M}^\#$ is defined as:

$$p \sqcap p' := \bigwedge_i(\phi_i) \bigsqcap \bigwedge_j(\phi'_j) := \bigwedge_{i,j}(\phi_i \sqcap {\phi'}_j)$$

where $\phi_i \sqcap {\phi'}_i := \phi_j$ if $\phi_i \equiv {\phi'}_j$ and $\top$ otherwise. "$\equiv$" is a custom equivalence relation that is slightly less strict than the syntactical equality and supports predicate commutativity. There are many possibilities to improve this semantical equality and keep as much information as possible. The idea behind using this slightly improved intersection to approximate the disjunction of conjunctions of predicates is that for any set A, B, $(A \cap B) \subseteq (A \cup B)$.

4.4 Infeasible path identification

We now have to exploit the information we have on the state of the program at the different control points to find infeasible paths. In order to do this, we use a SMT solver, that is, a SAT solver enhanced with several theories including integers. This kind of tool allows us to find inconsistencies in our list of predicates via its C++ API. It may return `SAT` when it is able to exhibit a solution to the problem, or `UNSAT`, what guarantees that the problem has no solution. The SMT solver is systematically called at the end of the analysis of each basic block to test the consistency of the predicate list.

We chose CVC4 [1] for its high performance at recent contests such as SMT-COMP [2], its very open license and its rich API. CVC4 also supports (although partially) "unsat cores", a technique that finds a minimal subset of predicates that caused unsatisfiability. For instance, for such a conjunction of predicates:

$$(x \neq 2) \wedge (y > z) \wedge (x = y + z) \wedge (z = 1) \wedge (y \leq 0)$$

the unsat core module of CVC4 would return $\{y > z,\ z = 1,\ y \leq 0\}$.

There may be several minimal unsatisfiable subsets of predicates, in this case CVC4 will return only one of them. This feature is still being worked on, and we have observed great improvements in the success rate of this feature in the recent development versions of this solver. Still, the interface with CVC4 is separate from the rest of our tool and it can be enhanced with the ability to call other SMT solvers.

Let ϕ_i be the set of predicates returned as an unsat core and E_{ϕ_i} the edges of each predicate ϕ_i. Any path traversing $F_p = \cup E_{\phi_i}$ should be infeasible, but because of side-effects on parallel paths, F_p may include actually feasible paths, so we only consider F_p valid when there is no feasible path traversing F_p (and we manually check for that). In the worst case, we will have to use a full, unminimized path, but this will not cause us to miss infeasible paths, only to get a bigger and less "factorized" output.

Table 1 Results on the Mälardalen benchmarks.

			Inf. paths found with minimization			w/o minimization
Benchmark	BB (#)	Time (s)	1 edge	Minimized	Non-minimized	Non-minimized
Small benchmarks (no merging required)						
ndes	57	0.267	0	0	0	0
expint	70	0.748	4	5	5	34
edn	75	0.537	2	0	0	2
prime	118	4.368	2	8	12	43
compress	122	1.801	2	8	0	19
select	136	45.598	0	4	0	8
qsortexam	155	28.201	2	4	2	12
adpcm	323	0.074	3	0	0	3
Large benchmarks (merging required)						
ud	153	17.477	3	3	0	23
minver	449	188.339	4	0	0	16
statemate	453	193.849	0	16	0	22
ludcmp	632	143.088	5	6	0	510
nsichneu	754	250.385	0	1352	3234	8620
qurt	2777	773.805	3	0	0	3
lms	3098	915.434	26	22	211	2376
fft1	6123	2223.125	0	25	0	815

5 Experimental Results

We have run our analysis program on the Mälardalen benchmarks, compiled into ARM binaries by `gcc -O1` (minimal optimizations) with non-constant global memory to ensure multiple paths, performed on a 2.90GHz i7-4600M CPU, 4GB memory. Measuring the efficiency of such an analysis is tough for multiple reasons: (a) we do not know how many actual infeasible paths the program contains, (b) since our analysis is targetted to exhibit infeasible paths, but not to exploit them, any measured improvement on the computed WCET estimation is also due to the efficiency of the tool that will exploit these infeasible paths, (c) our infeasible paths are actually sets of infeasible paths, and we do not know how many paths they include. Even if we stop trying to find minimal sets of edges, the analysis cuts paths once they are identified as infeasible, and we do not know how many paths have been cut this way.

Table 1 showcases both the performance of the analysis on this set of benchmark and the powerfulness of the minimization algorithm. The benchmarks are sorted by number of basic blocks (after virtual inlining of function calls). The 17 smallest benchmarks with a size ranging from 2 to 50 basic blocks have been removed, 10 of which gave no results.

The first three result columns show the amount of unique infeasible paths found by the analysis split in three kinds: paths made of only one edge, successfully minimized paths, and paths for which the minimization failed. The last column gives the amount of infeasible paths found when path minimization is disabled. For example, running on “prime” with minimization, the analysis finds $2 + 8 + 12 = 22$ infeasible paths, including 10 successfully minimized ones. These 10 short-length paths translate into $43 - 12 = 31$ lengthy infeasible paths when no minimization work is done. “fft1” is an extreme: 815 paths have been minimized into 25 paths of an average length of 2.84 edges. We have observed that most of the computing time is spent on SMT solving.

The results look very promising for the bigger benchmarks and the occasional merges of predicates seem enough to tone down the combinatory explosion without hurting results too much, yet the actual impact on the precision of the computed WCET remains unknown.

6 Conclusion and Future Works

This article proposes a new approach to discover infeasible paths in a binary program. Our solution is a static analysis of (a) a CFG whose blocks are made of machine instructions (abstracted by semantic instructions) and (b) of program data states represented by predicates on registers and memories. Unsatisfiability by SMT of predicates allows identifying infeasible paths. The result is a list of edges of the program CFG that are forbidden on a feasible execution path. This information is typically used to tighten the precision of the WCET computation.

Although the proposed approach gives promising results, we feel that some infeasible paths remain undiscovered because of (a) too coarse states join operator and (b) of time calculation explosion. Issue (a) is the most important although it means that we have to find smart fixpoints for our predicates. Polyhedra could be an efficient and well-known alternative but we would be bound to linear predicates.

The second issue may be solved by reducing the amount of produced predicates. For example, code slicing would be a good technique to keep only code involved in conditions and therefore to produce only predicates involved in conditions. As a significant part of analysis time is spent in SMT solving, another solution might be to reduce the number of calls to the solver. We have to find a tradeoff during fixpoint computations between the frequency of SMT calls and the minimization of found paths: an SMT call finds infeasible paths but also allows reducing the number of states.

Finally, the infeasible paths found by our approach only concern mutual exclusivity between CFG edges. Our feeling is that quantitative relations, induced by the program semantics, exist between CFG edges and we wish to investigate this domain deeper afterwards.

References

1 C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

2 C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

3 H. Cassé, F. Birée, and P. Sainrat. Multi-architecture value analysis for machine code. In *WCET'13*, pages 42–52. OASICs, Dagstuhl Publishing, 2013.

4 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. *WCET*, 15:136–146, 2010.

5 J. Henry, M. Asavoae, D. Monniaux, and C. Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. *SIGPLAN Not.*, 49(5):43–52, June 2014.

6 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting C. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 358–363, 2006.

7 N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing – EDCC 5*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.