20th International Workshop on Worst-Case Execution Time Analysis

WCET 2022, July 5, 2022, Modena, Italy

Edited by

Clément Ballabriga

OASICS

Editors

Clément Ballabriga
Lille University, France
clement.ballabriga@univ-lille.fr

ACM Classification 2012
Computer systems organization → Real-time systems; Theory of computation → Program analysis; Software and its engineering → Software verification and validation; Software and its engineering → Software safety; Software and its engineering → Software performance

ISBN 978-3-95977-244-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-244-0.

Publication date
July, 2022

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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.WCET.2022.0

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface

Welcome to the 20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022). WCET 2022 is organized in conjunction with the Euromicro Conference on Real-Time Systems (ECRTS 2022) in Modena, Italy, and is held the first day of the conference (July 5th). The WCET workshop is the main venue for research on worst-case execution time analysis in the broad sense and serves as a yearly meeting for the WCET community.

This year, the workshop features an invited keynote talk by Prof. Peter Puschner from Technische Universität Wien entitled *From Timing Prediction to Predictable Timing* and 3 presentations of regular papers. Each of these 3 papers received 3 reviews from members of the program committee. The final selection was then based on an online discussion.

The WCET workshop is the result of the combined effort of many people. First, I like to thank the authors of the WCET 2022 papers, and the keynote speaker Prof. Peter Puschner, for contributing the scientific content of the workshop. I also thank the members of the program committee for their high-quality reviews and fruitful online discussion. I thank the steering committee for their guidance and advice on the organization of this workshop. Special thanks go to Michael Wagner for the help in publishing the proceedings of WCET 2022.

The WCET exists to exchange ideas on all WCET-related topics in a friendly atmosphere. I invite you to enjoy the presentations and to actively participate in the discussions!

Lille, France
June 28th, 2022
Clément Ballabriga

Committees

Program Chair

- Clément Ballabriga – Lille 1 University, France

Program Committee

- Jan Reineke – Saarland University, Germany
- Enrico Mezzetti – Barcelona Supercomputing Center, Spain
- Pascal Sotin – Université de Toulouse - IRIT, France
- Pascal Raymond – VERIMAG/CNRS, France
- Florian Brandner – Télécom Paris, France
- Martin Schoeberl – Technical University of Denmark, Denmark
- Renato Mancuso – Boston University, USA
- Björn Lisper – Mälardalen University, Sweden
- Heiko Falk – Hamburg University of Technology, Germany
- Jakob Zwirchmayr – TTTech Auto, Austria
- Damien Hardy – IRISA, France
- Peter Wägemann – Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Steering Committee

- Björn Lisper – Mälardalen University, Sweden
- Isabelle Puaut – University of Rennes I/IRISA, France
- Jan Reineke – Saarland University, Germany

StAMP: Static Analysis of Memory Access Profiles for Real-Time Tasks

Théo Degioanni
École Normale Supérieure de Rennes, France

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

Accesses to shared resources in multi-core systems raise predictability issues. The delay in accessing a resource for a task executing on a core depends on concurrent resource sharing from tasks executing on the other cores. In this paper, we present StAMP, a compiler technique that splits the code of tasks into a sequence of code intervals intervals, each with a distinct worst-case memory access profile. The intervals identified by StAMP can serve as inputs to scheduling techniques for a tight calculation of worst-case delays of memory accesses. The provided information can also ease the design of mechanisms that avoid and/or control interference between tasks at run-time. An important feature of StAMP compared to related work lies in its ability to link back time intervals to unique locations in the code of tasks, allowing easy implementation of elaborate run-time decisions related to interference management.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems

Keywords and phrases Worst-Case Execution Time Estimation, Static Analysis, Multicore, Interference, Implicit Path Enumeration Technique

Digital Object Identifier 10.4230/OASIcs.WCET.2022.1

Supplementary Material *Text (Appendix)*: `https://files.inria.fr/pacap/puaut/papers/WCET_2022_appendix.pdf`

Acknowledgements The authors would like to thank Abderaouf Nassim Amalou and the anonymous reviewers for their fruitful comments on earlier drafts of this paper.

1 Introduction

In order to guarantee timing constraints of real-time software, an upper bound of the Worst-Case Execution Time (WCET) of its sequential tasks is needed [17]. Static WCET estimation techniques provide such upper bounds (WCET *estimates*) and are well understood for single-core processors. However, multi-core architectures are now commonplace as they offer unprecedented processing power and low power consumption. Applying static WCET analysis to multi-core systems is more difficult than single-core ones since a task running on a core may suffer from interference delays caused by resource sharing with software executing on the other cores. Shared resources may be the Last-Level Cache (LLC), the memory bus or the memory controller.

Different approaches may be used to deal with interferences (see [11] for a survey). One class of techniques is to avoid interferences, by using, for instance, specific task models like PREM (PRedictable Execution Model, [13]), which separates the code of each task in a memory phase and an execution phase that does not perform any memory access. Specific scheduling techniques can then be designed to avoid the co-scheduling of phases that interfere with each other [14,18]. Another class of approaches allows interferences to occur at run-time, and leverages knowledge of the usage of shared resources to compute the resulting worst-case interference delays [7,9]. This latter class of approaches relies on knowledge of shared resource usage of tasks executing concurrently, but surprisingly few methods provide such information at a granularity smaller than the entire task.

In this paper, we propose StAMP, a static analysis technique that divides the code of a compiled program into a *linear sequence of non-overlapping code intervals*, each with a distinct worst-case memory access profile. State-of-the-art methods for the static analysis of memory accesses [2,12] all rely on *time-based* approaches: they divide the execution timeline into *time segments* for which they compute a worst-case number of memory accesses. In these approaches, there is no direct way to link segments back to concrete code sections. Therefore, run-time decisions to avoid or control interference have to rely on time only and are agnostic to the code location where the decision is taken. As compared with these time-based techniques, StAMP can link back its output time segments to *concrete code intervals*. This allows run-time decisions to be based not only on time but also on the code interval under execution. Such run-time decisions may, for instance, re-calculate interference based on the current progress of tasks or introduce synchronizations to avoid or minimize interference between intervals [15,16].

Dividing the binary code of tasks into a linear sequence of code intervals is based in StAMP on a compiler technique that operates at the binary level. StAMP first divides the binary code into a tree of "well-formed" regions, called Single Entry point Single Exit point (SESE) regions [8]. The advantage of using such regions is that the entry and the exit of each region are natural frontiers for intervals, and as such natural points for introducing interference-related scheduling decisions. StAMP generates different sizes of intervals depending on the depth at which the SESE region tree is explored (deeper exploration induces finer-grain intervals). It is then possible to apply worst-case memory access analysis to each interval individually.

Traditional extraction of SESE regions is edge-centric [8], creating sections with a single entry edge and a single exit edge. We show in this paper that when using node-centric SESE regions (regions with a single entry node and a single exit node), the number of regions is more important than when using edge-centric regions. This provides fine-grain regions to scheduling strategies, in particular on code with many branches, which we believe will improve the quality of scheduling strategies.

The contributions of this paper are the following:

- We propose StAMP, a compiler technique that splits the binary code of a task into consecutive code intervals. Similarly to state-of-the-art techniques [2,12], StAMP generates worst-case memory access profiles for intervals with known WCETs. However, in contrast to [12] and [2], StAMP links back intervals to locations in the code of tasks. Moreover, the algorithmic complexity of StAMP is much lower than the one of the algorithms from [2].
- We provide an extensive experimental evaluation of StAMP, showing in particular that:
  - Interval extraction using node-centric SESE regions results in finer-grain regions than traditional edge-centric regions.
  - Controlling the depth at which the SESE region tree is explored allows us to control the size of the produced intervals.

The memory access profiles generated by StAMP can be used by off-line scheduling strategies to minimize interference overhead. Moreover, the fact that StAMP links back intervals to locations in the code of tasks provide useful information to take elaborate run-time decisions such as dynamic reconfiguration of schedules and re-calculation of interference delays [15,16]. This paper focuses on calculation of memory access profiles, their use for off-line or on-line scheduling strategies is considered outside the scope of the paper.

The rest of this paper is organized as follows. The method implemented in StAMP is described in detail in Section 2. Experimental results are given in Section 3. Section 4 compares our approach to related techniques. We finally discuss the results achieved and present our future work in Section 5.

2 Estimation of memory access profiles with StAMP

After presenting the system model StAMP relies on (Section 2.1, the properties of code intervals as computed by StAMP are detailed in 2.2. SESE regions, the blocks from which code intervals are constructed, are defined in 2.3. Sections 2.4 and 2.5 then respectively present the construction of code intervals and the calculation of their worst-case number of memory accesses.

2.1 System model and problem statement

StAMP operates of the binary code of an individual task, from static analysis of its Control Flow Graph (CFG). The target architecture may have a complex memory hierarchy (instruction and data caches). We assume that there exists a way (for instance static cache analysis as in our experimental evaluation in Section 3) to figure out if an access to a given address may result in a memory access.

The problem addressed by StAMP is the following. Given the code of a task, StAMP splits its code in consecutive code intervals (formally defined in 2.2) and for each of them calculates the worst-case number of memory accesses that may occur when executing the interval (*memory access profile*).

2.2 Code intervals

The analysis in StAMP first divides a given control-flow graph into consecutive *code intervals*, each linked to a code section, for which we individually compute the number of worst-case memory accesses. Then, WCET analysis is performed to convert the code-based segmentation into a time-based segmentation, allowing a straightforward bidirectional mapping between the two.

Picking the right abstraction to represent code intervals is critical for the correctness and effectiveness of the method. In this work, a code interval is a single-entry, single-exit sub-graph of the control-flow graph. Code intervals cover the entirety of the control-flow graph and are chained as a sequence. Figure 1 represents an example control-flow graph along with an example code interval cover for it.

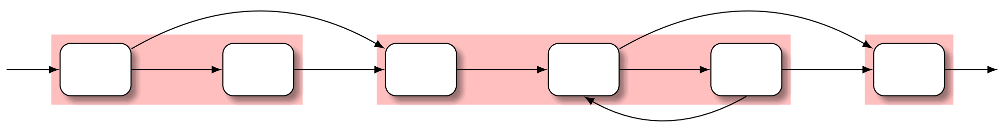

Figure 1 Example control-flow graph in black with an example code interval cover of length 3 in pink. Notice how all exit edges of a given code interval point to the entry point of the next code interval.

More formally, code intervals are defined as follows (Definition 3), based on the concepts of Control Flow Graph (Definition 1) and Code Interval Cover (Definition 2).

▶ **Definition 1** (Control Flow Graph (CFG)). *A CFG is a directed connected graph $C = (V, E)$, with V the vertices of the CFG (basic blocks, straight-line sequences of instructions with no branch no out except at the exit) and E the edges (pair of nodes, subset of $V \times V$), that represent the possible control flows between basic blocks. A CFG can contain special vertices (call nodes) representing calls to another CFG. A CFG has a single entry node and a single exit node.*

▶ **Definition 2** (Code interval cover). *Let $n \in \mathbb{N}$. A code interval cover of length n is a partition $(I_k)_{k\in\{1,\dots,n\}}$ of a CFG C such that for all $k \in \{1, \dots, n\}$, the following properties hold:*

- *If there exists $m \in \{1, \dots, n\}$ different from k such that there exists $(v_1, v_2) \in E$ with $v_1 \in I_k$ and $v_2 \in I_m$, then m and v_2 are unique.*
- *No such m exists if and only if I_k contains the exit node of the CFG.*

▶ **Definition 3** (Code interval). *A code interval is an element of a code interval cover.*

A direct consequence of Definition 2 is that a code interval cover always covers the entirety of the CFG, as it is a partition. A node of the CFG is thus always part of exactly one code interval. Another consequence is that the exit edges of a given code interval always enter the same code interval, as m in the definition is unique, and only the interval containing the exit node does not have exiting edges. A code interval cover is thus always a *chain* of code intervals. Note that a code interval cover always exists for a given control-flow graph, as the single-element partition of the control-flow graph itself is a valid code interval cover.

2.3 SESE regions

The single-entry and single-exit nature of code intervals invites us to formally introduce and use the notion of Single Entry Single Exit (SESE) regions. SESE regions may be edge-centric as originally introduced in [8] or node-centric.

▶ **Definition 4** (Edge-centric SESE region). *An edge-centric SESE region of a CFG $C = (V, E)$ is a subset $R \subseteq V$ for which there exists $e_{in} \in E$ and $e_{out} \in E$ such that:*

- *For all $(v_1, v_2) \in E$, if $v_2 \in R$ then either $v_1 \in R$ or $(v_1, v_2) = e_{in}$.*
- *For all $(v_1, v_2) \in E$, if $v_1 \in R$ then either $v_2 \in R$ or $(v_1, v_2) = e_{out}$.*

Previous work has shown that it is possible to generate edge-centric SESE regions in a tree arrangement in linear time [8]. A parent-child relationship in the tree indicates that the child region is completely nested in its parent, and detected regions never partially overlap. An example of edge-centric tree-arranged SESE regions (from the CFG of Figure 1) is represented in Figure 2.

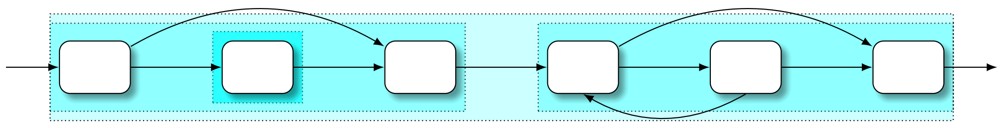

Figure 2 Control-flow graph from Figure 1 with stacked non-overlapping edge-centric SESE regions in blue (darker is deeper in the SESE region tree).

As a side-product of our approach, we propose another way to define SESE regions. Instead of defining frontiers of regions as entry and exit *edges*, we focus on entry and exit *nodes*. Similarly to edge-centric regions, node-centric regions can be arranged in an inclusion tree. An example of node-centric tree-arranged SESE regions (from the CFG of Figure 1) is represented in Figure 3.

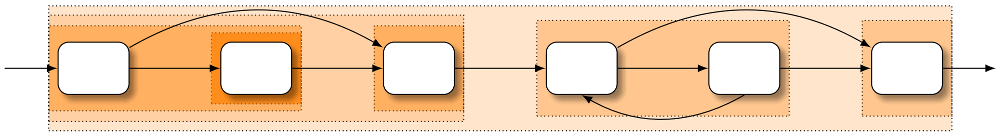

Figure 3 Control-flow graph from Figure 1 with stacked non-overlapping node-centric SESE regions in orange (darker is deeper in the SESE region tree).

▶ **Definition 5** (Node-centric SESE region). *A node-centric SESE region of a CFG $C = (V, E)$ is a subset $R \subseteq V$ for which there exists $v_{in} \in V$ and $v_{out} \in V$ such that:*

- *For all $(v_1, v_2) \in E$, if $v_2 \in R$ then either $v_1 \in R$ or $v_1 = v_{in}$.*
- *For all $(v_1, v_2) \in E$, if $v_1 \in R$ then either $v_2 \in R$ or $v_2 = v_{out}$.*

Node-centric regions are extracted from the CFG of a program using Algorithm 1. A node-centric SESE region is canonically represented by its pair (v_{in}, v_{out}).

Algorithm 1 Algorithm to compute node-centric SESE regions from a control-flow graph C. Returns a set of regions that are represented as their pair (v_{in}, v_{out}). This algorithm has a worst-case time complexity of $O(n^3)$ where n is the number of CFG nodes in a function.

```
function ComputeNodeSESE($C$)
    compute dominators and post-dominators of $C$
    regions := {}                                   ▷ Generate all regions (including overlap).
    for each node $start \in V$ do                                              ▷ $C = (V, E)$
        for each dominator $end$ of $start$ do
            if $start$ is a post-dominator of $end$ then
                $inside$ := nodes in region ($start$, $end$) ignoring back edges
                if all back edges in region ($start$, $end$) link two nodes of $inside$ then
                    regions.push(($start$, $end$))
                end if
            end if
        end for
    end for
    overlapping_regions := {}                                ▷ Remove overlapping regions.
    for each $r_1$ in regions do
        for each $r_2$ in regions do
            if $r_1 \cap r_2 \neq \emptyset$ and $r_1 \not\subseteq r_2$ and $r_2 \not\subseteq r_1$ then
                overlapping_regions.insert($r_1$)
                overlapping_regions.insert($r_2$)
            end if
        end for
    end for
    return regions $\setminus$ overlapping_regions
end function
```

Algorithm 1 is based on the well-know concepts of dominators and post dominators used in compilers[1]. It is divided in two loops: one generating all node-centric SESE regions, that could possibly overlap, and one filtering out overlapping regions to enforce the inclusion property among regions. The second loop operates as follows. Consider two regions $r_1 = (n_1, m_1)$ and $r_2 = (n_2, m_2)$ that overlap without inclusion. We can consider without loss of generality that

[1] A node d *dominates* a node n if every path from the entry node of the CFG to n must go through d. A node d *post-dominates* a node n if every path from n to the exit node of the CFG must go through d

n_2 is also a node of r_1. Therefore, r_1 and r_2 contain three node-centric SESE regions that do not overlap: (n_1, n_2), (n_2, m_1) and (m_1, m_2). As these three regions cover the exact same nodes as r_1 and r_2, r_1 and r_2 can be filtered out.

2.4 Computing code interval covers

Code interval covers can be computed through a depth-first traversal over the SESE tree. The concept of SESE tree is defined in Definition 6. Whether the kind of regions used is edge-centric or node-centric does not influence the definition.

▶ **Definition 6** (SESE tree). *A SESE tree has two constructors:*

- BasicBlock(n) *containing a control-flow graph with node n alone.*
- Region(r, *children*) *containing a SESE region r and a non-empty set of SESE tree children. The two following properties must also hold:*
  - *Any CFG node covered by an element of children must be covered by r.*
  - *Any CFG node in r must be covered by exactly one element of children.*

From this tree definition, we derive Algorithm 2 to generate a code interval cover.

Algorithm 2 Computes a control-flow ordered code interval cover, taking as parameters a SESE tree and a fuel amount. The code interval cover is represented as a sequence of SESE trees exactly covering the nodes of the code interval. The fuel parameter controls the exploration depth: the higher the fuel, the more fine-grain the cover.

```
function CodeIntervalCover($tree$, $fuel$)
    if $tree$ is a BasicBlock($n$) then
        if $n$ is a call node to $callee$ then
            return [$tree$] + CodeIntervalCover($callee$, $fuel$)
        else if $n$ has up to one control-flow successor then
            return [$tree$]
        else
            return Unchainable
        end if
    else if $tree$ is a Region($r$, $children$) then
        if $fuel = 0$ then
            return [$tree$]
        else                ▷ Build CFG ordering of SESE children of $tree$, if possible.
            $result\_intervals$ := [ ]
            $child$ := control-flow entry of $children$
            while $child \neq null$ do
                $child\_result$ := CodeIntervalCover($child$, $fuel - 1$)
                if $child\_result$ = Unchainable then
                    return [$tree$]
                end if
                $result\_intervals := result\_intervals + child\_result$
                $child$ := any control-flow successor of $child$ in $children$
            end while
            return $result\_intervals$
        end if
    end if
end function
```

This algorithm takes as parameters a SESE tree *tree* and an amount of *fuel* modeling the maximum depth of the recursive exploration of *tree* (each recursive call consumes one unit of *fuel*), and returns a sequence of SESE trees that will each represent a code interval. The depth-first traversal matches on the two constructors of SESE trees recursively, reducing *fuel* on each recursive call. In the BasicBlock case, we either forward the construction to a called CFG if there is one by unfolding the CFG of the callee, or check whether the node can be part of a chain (returning Unchainable if not). In the Region case, we either end the exploration if there is no fuel left, or recursively call the construction function on the SESE children of the region, in control-flow order. Note that if one of the children cannot be chained, we abort breaking the region in parts and simply return the region itself, as we could not output a more precise code interval. The worst-case time complexity of the algorithm is linear with respect to the number of SESE tree nodes, which in the general case is equivalent to quadratic with respect to the number of CFG nodes in a function.

2.5 Computing memory access profiles

Once the control-flow graph is divided into code intervals, we compute the worst-case number of memory accesses (WCMA) for each interval by largely relying on standard Implicit Path Enumeration Technique (IPET) analysis [10], both for estimating: (i) the (partial) WCET of code intervals; (ii) their worst-case number of memory accesses (WCMA).

To compute the WCET of each code interval, we constrain to zero the WCET value of each basic block outside the code interval under analysis and outside any of the functions it could call (recursively). The WCMA of each code interval is estimated in a similar way, by setting to zero the WCMA of each basic block outside the interval under analysis. This partial WCET/WCMA calculation is straightforward due to the single-entry single-exit feature of code intervals.

Detecting if a load/store instruction may result in memory access depends on the presence of instruction/data caches in the architecture under analysis. The experimental evaluation of StAMP uses an architecture with instruction and data caches, thus not all load/store instructions result in memory accesses, as explained in Section 3.1.

Then, the only step left is to create a memory access profile out of the code intervals.

▶ **Definition 7** (Memory access profile). *A memory access profile is a sequence of pairs* $(wcet, wcma)$ *representing a time-sequence of code intervals of maximum duration wcet in which at worst wcma memory accesses can happen.*

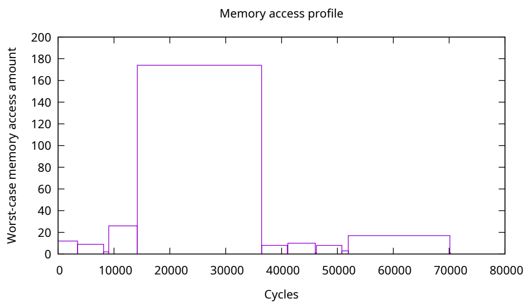

Figure 4 Example memory access profile (`minver`, node-centric SESE regions, non-limiting large value of fuel parameter.

As code intervals are chained in a sequence, it is possible to create a memory access profile by mapping code intervals in the produced sequence of code intervals to their pair $(wcet, wcma)$. An example of a memory access profile is given in Figure 4. The x-axis represents the code intervals with their WCET in cycles, while the y-axis represents the corresponding WCMA.

3 Experimental evaluation

In this section, we present the details of our implementation and discuss the benefits and limitations of StAMP through experiments.

3.1 Implementation of StAMP and experimental setup

StAMP was implemented within the Heptane WCET analysis platform [6]. In order to evaluate the quality of the generated memory access profiles, we ran StAMP on the benchmarks provided by Heptane (C code of the Mälardalen benchmarks [5] with loop bounds annotated using the Heptane format). The target architecture used is based on MIPS with a single layer of data cache and instruction cache. Both caches are 2-associative LRU caches with 32-bytes cache lines and 32 sets. Heptane provides a built-in instruction cache, data cache and address analysis. Every access not classified as *always-hit* by the cache analysis of Heptane is assumed to perform a memory access. This allows obtaining the Worst-Case number of Memory Access (WCMA) of each basic block. Heptane additionally provides IPET analysis for WCET computation, that was modified to compute the worst-case number of memory accesses of intervals. As such, most of the implementation work was to compute the code interval covers.

3.2 Memory access profile results

We ran StAMP on all benchmarks with a very large value for the fuel parameter (simply termed *unlimited fuel* hereafter). All the produced profiles are provided as supplementary material that can be downloaded from [4]. An example generated profile is given in Figure 4. Note that some intervals are only a couple of cycles long and are thus not visible on the graphical representation. While this graphical representation of memory access profiles is useful to compare StAMP with state-of-the-art methods, it does not translate all the capabilities of StAMP. Indeed, each block in the memory profile corresponds to a code interval. This allows mapping back each block's WCET and WCMA information (in the time domain) to a code interval (in the code domain). This is especially useful for schedulers which operate at run-time when the current position in the code is also known.

3.3 Granularity using edge-centric versus node-centric SESE regions

StAMP is generic over the flavor of SESE regions (edge-centric vs. node-centric) used to compute code intervals. Table 1 compares the length of code interval covers with edge-centric and node-centric regions.

With unlimited fuel, node-centric SESE regions systematically outperform edge-centric SESE regions in terms of length of the generated cover (the larger the number of intervals, the more precise the information provided to the scheduler). This can be explained by noticing that any edge-centric SESE region with $e_{in} = (s, t)$ and $e_{out} = (s', t')$ induces a node-centric SESE region with $v_{in} = t$ and $v_{out} = t'$. As such, there are always more node-centric SESE regions than edge-centric SESE regions.

One of the most interesting results is observed on benchmark `nsichneu`, that features many `if` statements. In this benchmark, edge-centric SESE regions generate only a single code interval, while node-centric SESE regions generate 127 code intervals. The node flavor of StAMP results in particularly fine-grain intervals, as shown in Figure 5. In this example, the large number of intervals may increase the complexity of off-line scheduling strategies. However, merging intervals is straightforward, because they form a sequence that covers all the code.

Table 1 Length (number of intervals) of code interval covers (CIC) (node vs edge-centric regions, unlimited fuel).

	CIC length			CIC length	
Benchmark	**Edge**	**Node**	**Benchmark**	**Edge**	**Node**
ud	11	**12**	select	**5**	**5**
insertsort	**3**	**3**	ns	3	**5**
sqrt	3	**4**	minver	**17**	**17**
matmult	**17**	**17**	crc	5	**15**
fibcall	3	**5**	minmax	1	**3**
fft	7	**9**	simple	1	**4**
cover	**13**	**13**	ludcmp	5	**6**
expint	3	**4**	qurt	7	**10**
jfdctint	**9**	**9**	bs	**5**	**5**
statemate	9	**17**	bsort100	7	**9**
lcdnum	**3**	**3**	nsichneu	1	**127**

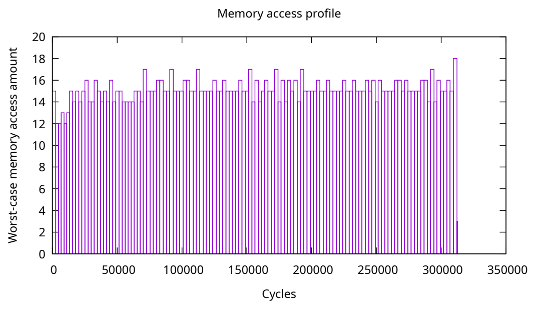

Figure 5 Memory access profile for benchmark `nsichneu` (node-centric regions, unlimited fuel).

3.4 Controlling the granularity of memory access profiles

The longer the generated code interval sequence length, the more accurate the interference calculation, but the more complex the scheduling, which motivates the need to control the length of the generated sequences. This control is provided in StAMP by the *fuel* parameter, which commands how deeply the SESE tree is traversed. Figure 6 illustrates the effect of varying the amount of fuel on the sequence length for node-centric regions. Evolution of `nsichneu` is not drawn for clarity as the curve goes very high. Its behavior is however similar to other benchmarks: after a few levels of recursion, it remains stable.

Augmenting the amount of fuel increases the generated code interval cover length. However, this control is limited, as SESE trees in practice are not very deep in our benchmarks. As seen in Figure 6, none of our benchmarks benefit from a value of fuel higher than five. Instead, granularity can be reduced by merging consecutive code intervals as their union forms a new code interval. This alternative method further allows control over the size of code intervals.

3.5 Limitation of code interval cover computation

The analysis in StAMP in some benchmarks does not go deep in the SESE tree no matter the amount of fuel, and the number of intervals detected is rather small. This phenomenon illustrates a limitation of the code interval model, occurring when control-flow bypasses a large section. As an example, Figure 7 shows a situation in which fine-grained intervals could not be generated because a bypass edge blocks the traversal of successors in Algorithm 2.

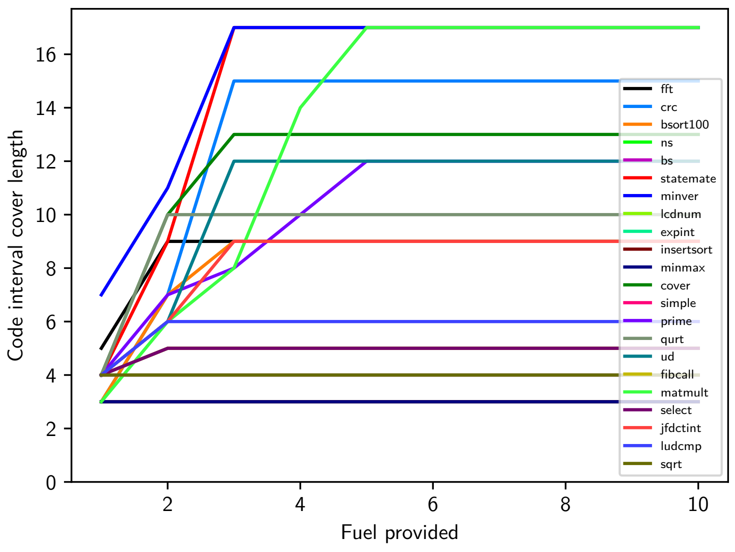

Figure 6 Influence of the amount of fuel on the code interval cover length (node-centric regions). `nsichneu` omitted for readability.

Some of our benchmarks (such as for example `expint` or `select`) resulted in a memory access profile containing a single dominant block (surrounded by negligibly short blocks). This is generally caused by `if` statements or loops enclosing most of the code as illustrated in Figure 7.

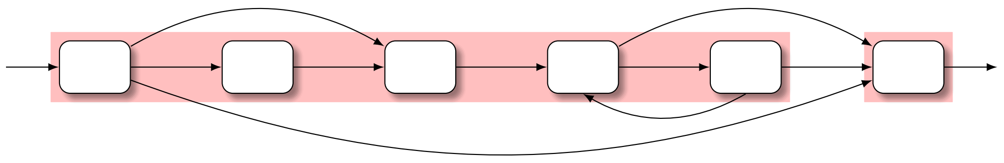

Figure 7 The same CFG as in Figure 1 is no longer dividable in a code interval cover longer than 2 if we add a bypass edge from the leftmost to the rightmost block. In red is the longest possible code interval cover.

4 Related work

The authors of [2] present TIPs, a technique to extract memory access profiles from code using trace enumeration. Similarly, Oehlert et al. present in [12] a technique to extract event arrival functions using IPET, with memory accesses as a particular case of intervals. These two papers generate memory access profiles per *interval*, with an interval defined as a time interval in task execution. StAMP, in contrast, first generates a profile in the code domain and then converts its results to the time domain. This allows introducing specific code (i.e. synchronization) for tighter identification of interference cost [15, 16].

The time-domain output of the TIPs method differs significantly from StAMP and the method presented by Oehlert et al. While the latter provides an upper bound of the number of memory accesses left to do after a certain point in time, TIPs provides more precise data that describe the worst-case number of memory accesses during the ongoing time interval. However, this is achieved via exponential time algorithms over the length of the program, while StAMP is in polynomial time over the length of the program.

The PREM (PRedictable Execution Model, [13]) allows to separate phases that use shared resources from those that do not and allows a scheduling technique that avoids the co-scheduling of phases that interfere with each other [14, 18]. Similarly to PREM, StAMP identifies code intervals with different memory access patterns to shared resources but offers more flexibility in the identified access patterns.

The MRSS task model introduced in [3] characterizes how much stress each task places on resources and how sensitive it is to such resource stress and presents schedulability tests using this model. Memory access profiles such as those produced by StAMP produce information at a finer granularity than in [3] (interval level instead of task level) and could be used to improve the schedulability tests of [3].

Code interval covers are similar to super blocks as defined in [1] in the sense that when an interval in the cover is executed, all other intervals will be executed exactly the same number of times, and as such enforce full coverage of the CFG. In contrast to the concept of super block as defined in [1] intervals in our sequences are made of sub-graphs of the CFG and not basic blocks.

5 Conclusion

We have presented StAMP, a technique that generates worst-case memory access profiles from compiled code. StAMP links bidirectionally time-domain and code-domain information. The technique generates a segmentation of a compiled program as a code interval cover in polynomial time and generates a time-domain representation of the worst-case number of memory accesses in this cover.

Future work should first experimentally evaluate the differences between the memory access profiles generated by StAMP and the ones obtained by time-based technique of [2], and most importantly their respective impact on scheduling strategies. Other directions for future work are to improve the expressiveness of the method by improving the code interval model (i.e., move from sequences of intervals to directed graphs), and to enhance memory access profiles to detail when memory accesses occur within code intervals, similarly to [12]. We also believe improvements to the worst-case complexity of the generation of non-overlapping node-centric SESE regions are possible.

References

1 Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 25–34, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.175935.

2 Thomas Carle and Hugues Cassé. Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. In Christian Hochberger, Lars Bauer, and Thilo Pionteck, editors, *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*, volume 12800 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2021. doi:10.1007/978-3-030-81682-7_2.

3 Robert I. Davis, David Griffin, and Iain Bate. Schedulability Analysis for Multi-Core Systems Accounting for Resource Stress and Sensitivity. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2021.7.

4 Théo Degioanni and Isabelle Puaut. Stamp: Static analysis of memory access profiles for real-time tasks: supplementary material, 2022. URL: `https://files.inria.fr/pacap/puaut/papers/WCET_2022_appendix.pdf`.

5 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASIcs*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. `doi:10.4230/OASIcs.WCET.2010.136`.

6 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In *International Workshop on WCET Analysis*, pages 8:1–8:12, 2017.

7 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2020.23`.

8 Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/178243.178258`.

9 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. `doi:10.1109/RTAS.2014.6925998`.

10 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC: 32nd ACM/IEEE conference on Design automation*, pages 456–461, 1995.

11 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. `doi:10.1145/3323212`.

12 Dominic Oehlert, Selma Saidi, and Heiko Falk. Compiler-based extraction of event arrival functions for real-time systems analysis. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 4:1–4:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ECRTS.2018.4`.

13 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011. `doi:10.1109/RTAS.2011.33`.

14 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst.*, 16(5s):164:1–164:20, 2017. `doi:10.1145/3126496`.

15 Stefanos Skalistis and Angeliki Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 233–245. IEEE, 2019. `doi:10.1109/RTSS46320.2019.00030`.

16 Stefanos Skalistis and Angeliki Kritikakou. Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2020.4`.

17 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008.

18 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real Time Syst.*, 48(6):681–715, 2012. doi:10.1007/s11241-012-9158-9.

LLVMTA: An LLVM-Based WCET Analysis Tool

Sebastian Hahn
Saarland University, Saarland Informatics Campus[1], Saarbrücken, Germany

Michael Jacobs
Saarland University, Saarland Informatics Campus[2], Saarbrücken, Germany

Nils Hölscher
TU Dortmund University, Germany

Kuan-Hsun Chen
University of Twente, The Netherlands

Jian-Jia Chen
TU Dortmund University, Germany

Jan Reineke
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We present LLVMTA, an academic WCET analysis tool based on the LLVM compiler infrastructure. It aims to enable the evaluation of novel WCET analysis approaches in a state-of-the-art analysis framework without dealing with the complexity of modeling real-world hardware architectures. We discuss the main design decisions and interfaces that allow to implement new analysis approaches. Finally, we highlight various existing research projects whose evaluation has been enabled by LLVMTA.

2012 ACM Subject Classification Software and its engineering → Software performance; Software and its engineering → Automated static analysis; Software and its engineering → Compilers; Computer systems organization → Real-time system architecture

Keywords and phrases WCET analysis, low-level analysis, LLVM

Digital Object Identifier 10.4230/OASIcs.WCET.2022.2

Supplementary Material LLVMTA and a patched version of LLVM are available as open source for academic research purposes:
Software (Source Code): `https://gitlab.cs.uni-saarland.de/reineke/llvmta`
Software (Source Code): `https://gitlab.cs.uni-saarland.de/reineke/llvm`

Funding This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS); by the Saarbrücken Graduate School of Computer Science, which received funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments; and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

Acknowledgements We would like to thank all the students at Saarland University who contributed during the development of LLVMTA: in particular Claus Faymonville for his extensive contributions in a very early development stage, but also Tina Jung, Darshit Shah, and Tobias Stark.

[1] Work underlying this paper was performed between 2014 and 2019 while the author was still working at Saarland University. Since 2019, the author is affiliated with AbsInt Angewandte Informatik GmbH which is not related to work described in this paper.
[2] Work underlying this paper was performed between 2014 and 2017 while the author was still working at Saarland University. Since 2018, the author is affiliated with Artengis GmbH which is not related to work described in this paper.

1 Introduction

In this paper, we introduce LLVMTA, an open-source worst-case execution time (WCET) analysis tool developed at Saarland University from 2014 onwards. Our aims with this paper are twofold: First, we want to convey the main design goals of LLVMTA and how these design goals manifest in its current design. Second, to facilitate future work on LLVMTA, we describe the most important interfaces, which need to be implemented to adapt and extend LLVMTA.

At the onset of the work on LLVMTA, four WCET analysis tools were at the disposal of the authors: the commercial WCET analyzer aiT by AbsInt GmbH, and the three academic WCET analysis tools OTAWA [7], developed in Toulouse, Chronos [49] developed in Singapore, and Heptane [38], developed in Rennes. Our goals at the time were to study WCET analysis for microarchitectures that feature timing anomalies [53, 64] and to explore the potential of compositional timing analysis [37]. In order to evaluate the full potential of the planned approaches, we decided to implement them in a state-of-the-art analysis framework. To the best of our knowledge, the required state-of-the-art analysis features (in particular abstract execution graphs, as discussed in Section 2) were only available in the commercial aiT tool created by Absint. Commercial analysis tools as aiT, however, exhibit a high degree of complexity as they have to support a wide range of real-world hardware platforms and need to scale to large real-world applications under analysis. Thus, we decided to create our own analysis framework from scratch, to enable the rapid prototyping of novel analysis approaches. Due to the use of state-of-the-art analysis techniques, it is still reasonable to judge whether the observed gain in precision and/or efficiency of novel analysis approaches would translate to commercial tools.

Also worth mentioning is the WCET compiler WCC [20] developed in Dortmund. WCC is a compiler focusing on WCET optimizations, implementing its own high- and machine-level intermediate representations, namely ICD-C and ICD-LLIR. WCC uses aiT for its timing analysis and does not provide timing analysis on its own. While LLVMTA is using the LLVM compiler infrastructure it does not perform code transformations on its own.

As part of T-CREST [60], a tool called Platin has been developed in Vienna. Platin also uses aiT to provide low-level timing analysis or, alternatively, an internal analyzer, which, however, does not feature a detailed microarchitectural analysis. Platin can compute loop bounds by both static analysis, using LLVM infrastructure, and by simulating short traces. These loop bounds can then be fed as flow facts to aiT.

We set out to create a new academic WCET analysis tool with these minimal requirements:

- Support of precise and accurate analysis of microarchitectures with timing anomalies using state-of-the-art techniques, in particular *abstract execution graphs* [72].
- Support of compositional analysis [37] approaches in which the analysis of different timing contributors is performed separately.
- Flexibility to easily replace pipeline or cache models or to add new path constraints.

WCET analysis is challenging for several reasons, including some that we were not particularly interested in, which entails some explicit “non-goals”:

- Our goals has been to study fundamental challenges in WCET analysis, not to support particular (commercial) microarchitectures. We note, however, that given sufficient knowledge of the underlying microarchitectures, it would be possible to support particular microarchitectures within LLVMTA.
- WCET analysis typically applies to binary executables. This entails the challenge of reconstructing the control-flow graph of the code, which is not explicit in the binary. LLVMTA bypasses this challenge by integrating into the LLVM compiler.

The remainder of this paper is structured as follows: We begin by giving a brief overview of the architecture of static WCET analyzers. Next, we present an overview of the architecture of LLVMTA. Subsequently, we present the usage of the command-line tool LLVMTA for a small example. Finally, we conclude the paper with a brief discussion of existing applications of LLVMTA by pointing out future steps in the development of the tool.

2 Standard Architecture of Static WCET Analysis Tools

In this section, we describe the *de facto* standard architecture underlying static WCET analysis tools today, which in particular underlies LLVMTA. There are fundamentally different WCET analysis approaches, such as measurement and hybrid WCET analysis, which are out of scope in this discussion.

LLVMTA and other WCET analyzers operate on a control-flow graph (CFG) representation of the program under analysis. A CFG is a directed graph whose nodes correspond to basic blocks, i.e. straight-line code sequences, and whose edges correspond to possible control flow between these nodes. The CFG of a program is not explicit in the machine code executed on the hardware. Thus the first step of most WCET analysis tools is to reconstruct a CFG from the program binary [75, 76, 45, 23, 67, 8]. In LLVMTA, the CFG is directly obtained from the compiler generating the binary rather than by reconstructing it.

Given the program's CFG, the WCET analysis problem can then be decomposed into three subproblems:

1. Deriving constraints that approximate the subset of paths through the control-flow graph that are semantically feasible.
2. Determining the possible execution times of program parts, such as basic blocks, accounting for the timing effects of microarchitectural features such as pipelining and caching.
3. Combining the information from 1. and 2. to derive a bound on the program's WCET.

The first subproblem depends only on the program's semantics and is thus independent of the underlying microarchitecture. If the program under analysis contains loops, it is necessary to bound each loop's maximum number of iterations; otherwise, no WCET bound can possibly be derived. Different *loop bound analyses* have been described in the literature [31, 52, 30, 74, 15, 19, 58, 6, 10]. Generalizing loop bound analysis, *control-flow analysis* derives constraints on the possible execution paths through the CFG [46, 16, 5, 42, 68, 59, 61] including loop bounds.

By definition, the second subproblem critically depends on the underlying microarchitecture. Thus the underlying analysis is often called *microarchitectural analysis*. Traditionally, the output of microarchitectural analysis have been bounds on the timing contributions of the basic blocks [7, 38, 49] of the program. As modern processors employ pipelining, the execution of successive basic blocks may overlap substantially. Thus, it is important not to "pay" for this overlap multiple times, and to analyze basic blocks within the context of their surrounding basic blocks. Different approaches to this end have been proposed [51, 77, 39, 53, 12, 22, 17, 78, 48, 65, 82]. We are unable to give a complete account of these approaches here due to space limitations; instead we focus on a brief description of the approach taken in LLVMTA.

Microarchitectural analysis in LLVMTA can be seen as a static cycle-by-cycle simulation of the execution of the program on *abstract* microarchitectural states, accounting for any microarchitectural component that influences the execution's timing, such as pipelining (including e.g. forwarding effects), branch prediction, load and store buffers, and caches.

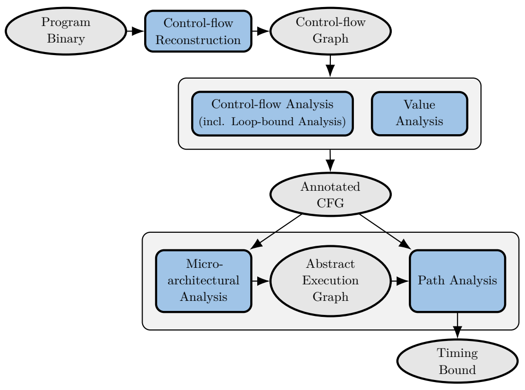

Figure 1 Overview of the general steps in WCET analysis.

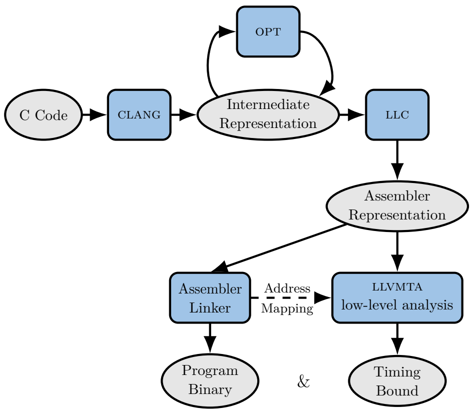

Figure 2 Overview of the common LLVM compilation flow (CLANG, OPT, LLC) including the integration of our low-level analysis tool LLVMTA.

Due to abstraction this static simulation may lack information, e.g. whether an access results in a cache hit or a cache miss, or whether two memory accesses alias, the simulation may have to "split" following multiple successor states, introducing nondeterminism. The output of microarchitectural analysis is an *abstract execution graph* (AEG) [72] whose nodes correspond to abstract microarchitectural states and whose edges correspond to the passage of processor cycles. An important distinguishing feature of this approach is that the AEG may capture correlations between the timing contributions of different basic blocks, rather than computing a single bound for each basic block. The key to make this approach successful in practice is to find abstractions that strike a good balance between analysis complexity and precision. For caches various compact abstractions have been developed [1, 25, 26, 27, 71, 33, 29, 11, 14, 28, 24, 54, 79, 9, 80] that offer varying degrees of precision depending on the underlying replacement policy [40, 63].

The third and final step of WCET analysis is to combine the information gathered in the first two steps to compute a bound on the program's WCET. The most popular approach to this *path analysis problem* is the *implicit path enumeration technique* (IPET) [50]. The basic idea behind IPET is to solve the path analysis problem via an integer linear program (ILP). Integer variables are introduced for each edge in the AEG, encoding the frequency of taking those edges during an execution. The structure of the AEG imposes linear constraints relating these frequencies, implicitly encoding all possible paths through the AEG. Additional constraints are obtained from control-flow analysis; otherwise unbounded solutions would be possible in the presence of loops. Finally the objective function captures the cost of a given path through the AEG. Maximizing the objective function yields a safe WCET bound provided that the previous analyses capture all possible executions of the program on the microarchitecture. The overall WCET analysis flow is depicted in Figure 1.

For a more detailed discussion of static WCET analysis and related techniques we refer to the survey paper by Wilhelm et al. [81]. The same techniques can be used to safely approximate the number of occurrences of other microarchitectural events [43]. E.g. one can similarly determine a bound on the number of cache misses in any possible execution of the program.

3 LLVMTA Tool Architecture

3.1 High-level Structure

We implemented a low-level analysis tool called LLVMTA, following the scheme sketched in Figure 1. In this section, we provide details on this tool. LLVMTA is based on the LLVM compiler infrastructure [47], and it is hooked into the common LLVM compilation flow as depicted in Figure 2.

Overall Tool Architecture. Given a C program, the compiler frontend CLANG (`https://clang.llvm.org`) translates the program into the LLVM intermediate representation. After an optional optimization phase (OPT), the program is further translated to the assembler code (LLC) which results in the final binary after the linking step. Our analyses are implemented on the final assembler representation in the LLVM backend which is the representation closest to the machine level. The timing bound determined by LLVMTA is valid for the resulting binary, i.e., it will change accordingly if the binary changes, e.g. due to different compiler optimizations.

The integration of low-level timing analysis and compilation offers several advantages. First, no control-flow reconstruction of the binary is required because control-flow elements such as functions, basic blocks, and loops are provided by the prior compilation step. Second, the low-level analysis in the backend can make use of (high-level) information obtained at earlier stages and maintained during compilation. On the downside, the analysis requires as input the program to be analyzed in LLVM intermediate representation, and provides timing estimates only for the binary produced by the specific compiler. Commonly, this representation can be obtained using the compiler frontend from the high-level source program, for example given in C. It is conceivable, but it has not been experimentally validated, to apply binary lifters [4, 2, 18] to obtain an intermediate representation directly from binaries. The analysis results would then be valid for binaries obtained by recompiling the intermediate representation. Furthermore, the addresses of the instructions and the static data are only known after the linking step and would have to be fed back to the low-level analysis for sound analysis results. This is currently not implemented, but there are no major technical obstacles to doing so.

LLVMTA low-level analysis. To obtain precise results, we have implemented *context-sensitive* analysis [70], i.e. the analysis distinguishes different contexts that influence the execution behaviour. As an example, the execution behaviour of the first iteration of a loop usually differs from the behaviour of later iterations because the caches are being filled during the first iteration [55]. To establish a context-sensitive analysis framework, we implemented trace partitioning [56] on the final assembler representation in the LLVM backend. Context sensitivity is achieved by partitioning the set of execution traces according to some predicate on traces. We implemented predicates to discriminate different iterations of a loop, as well as different call sites of a function. The degree of context sensitivity, i.e. the number and size of these predicates, is an analysis parameter.

Based on our context-sensitive analysis framework, we have implemented a value analysis that tracks constant values of registers and memory cells. This value information is used to derive address information for data accesses. Despite the simplicity of the analysis domain, it is sufficient to precisely analyse stack-relative accesses. For accesses to globally defined objects such as global arrays, our tool uses information provided by the compiler to determine the range of possible addresses.

In order to derive loop bounds, we use the LLVM-internal scalar evolution analysis that provides an upper bound on the iteration count of loops in their intermediate representation. Our tool matches loops in the assembler representation to loops in intermediate representation in order to automatically obtain upper loop bounds on the assembler level. Manual loop annotations can be provided by the user for loops with complex iteration patterns. The scalar evolution analysis, originally based on [6] and extended in [10], computes a closed-form expression to describe how the values of variables evolve within a single loop iteration. These expressions are used to derive upper loop bounds, either in the form of numeric values or symbolic expressions w.r.t. the function parameters.

Our tool supports the analysis of different generic hardware platforms rather than proprietary industrial platforms for the ARM and the RISC-V instruction sets. This is sufficient to evaluate the general concepts used in timing analysis and takes significantly less effort to implement. We model textbook pipelines (see [41]) with in-order, strictly in-order [35, 32, 36], and out-of-order execution. The microarchitectural analysis supports scratchpad memories, as well as caches with least-recently-used replacement policy and both write-through and write-back policy. We have implemented must, may, and persistence cache analysis [1, 54, 62]. As background memory, the tool supports fixed-latency memory as well as dynamic random-access memory with a closed-page controller and distributed refreshes.

LLVMTA implements the *fast-forwarding* technique presented in [44] to increase the performance of the microarchitectural analysis. This optimization exploits the fact that pipelines tend to *converge* while waiting for memory, i.e. the pipeline cannot advance further until the current memory request is finished. Once converged, the (abstract) state of the pipeline stays the same as long as the memory is busy.

The abstract execution graph produced by the microarchitectural analysis is compressed afterwards. LLVMTA supports two different levels of compression. Either all start and end nodes within a basic block are kept separate to allow for a precise path analysis [72], or the graph is compressed into a single edge per basic block to allow for an efficient path analysis. Our tool supports multiple solvers to solve the ILP formulation resulting from the path analysis, including the commercial tools IBM ILOG CPLEX Optimization Studio (`https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex`) and Gurobi Optimizer (`https://www.gurobi.com`) that exhibit the best performance [57].

3.2 Limitations

While offering many advantages such as code reuse and flexibility, the nature of an academic prototype and the tight coupling with a compiler infrastructure also comes with limitations.

LLVMTA operates on the machine-level IR rather than on the binary, which may yield results that are not entirely faithful to the generated machine code for the following reasons. The assembler may break down pseudo-assembly instructions used in the machine-level IR – the level we perform the analysis on – into several machine instructions in the actual binary (especially on RISC-V). The address mapping is only determined after linking, and LLVMTA currently operates on a made-up address mapping. We note that it would be possible to obtain a faithful address mapping from the linker. For the ARM instruction set, predication is supported for branch instructions, but not for arbitrary machine instructions as specified in the instruction set architecture. As mentioned earlier, there are currently no microarchitectural models that correspond to existing commercial hardware designs. Finally, the tool is reasonable fast on the standard WCET benchmarks, but will likely not scale to real-world applications.

3.3 Main Design Interfaces

To reach our goal of flexibility, we use shared interfaces. New low-level analyses can be obtained by implementing these interfaces with new classes (possibly inheriting existing ones) and directing the analysis framework to employ these implementations. The most important interfaces of LLVMTA allow for:

- static program analysis on machine-level LLVM intermediate representation,
- microarchitectural analysis, in particular pipeline modeling,
- cache analysis, and
- additional path analysis constraints.

3.3.1 Program Analysis at Machine-level Intermediate Representation

The interface class `ContextAwareAnalysisDomain` enables context-sensitive analysis on a control-flow graph with machine-level instructions.

Part of the interface specifies the basic operation on abstract domain values in the spirit of abstract interpretation [13]:

- `isBottom`: does the current abstract value represent the bottom element of the analysis lattice?
- `lessequal` compares the current abstract value with another given one w.r.t. the partial order $\sqsubseteq$ of the analysis domain,
- `join` joins a given abstract value into the current abstract value w.r.t. the partial order of the analysis domain. The behaviour should be consistent with `lessequal`.

The second part of interface specifies the transfer behaviour of abstract values while abstractly interpreting the control-flow graph of the program under analysis.

- `transfer` takes the next instruction to analyze, the current analysis context, and, optionally, analysis information of preceding static analyses at this program point. It modified the current abstract value by the effect of the instruction in the specified context.
- `guard` can be used to sharpen the current abstract value by the knowledge of the outcome of a branch instruction (either taken or not taken).
- `enterBasicBlock` models the effect of entering a basic block on the analysis information.

3.3.2 Microarchitectural Analysis

The interface class `MicroArchitecturalState` models the abstract state of the microarchitecture under analysis. Microarchitectural analysis, unlike most program analysis techniques operates at the granularity of processor cycles rather than program instructions. `MicroArchitecturalState` has the following interface:

- The constructor of the class creates the initial microarchitectural state from which the state space exploration starts. This usually represents a state with an empty pipeline and unknown cache contents.
- `cycle` models the behavior of executing the machine for a single cycle. The abstract microarchitectural state is modified in-place. It takes a configurable set of precomputed analysis information, e.g. address information for memory-accessing instructions.
- `isFinal` specifies whether a given instruction has just finished execution in the current microarchitectural state. An instruction is hereby identified by its instruction address and a context. This predicate allows use to map microarchitectural states to instructions in the control-flow graph of the program under analysis. This is mostly a technicality, but influences e.g. at which points during the analysis microarchitectural states can be *joined*.
- `isJoinable` and `join` are used to test whether microarchitectural states can be joined - and do so if it is possible.

The interface class also features helper functions to model common functionality found in any microarchitecture such as the program counter and its evolution during program execution.

The `cycle` behaviour implicitly induces an microarchitectural state graph from the initial state up to states in which the last instruction of the program under analysis finished execution. At cycle granularity, i.e. having one edge per single execution cycle, such graphs are too large to be used in path analysis. In Section 3.3.4 below, we will describe how to obtain a more compact graph where edges are collapsed to describe multiple execution cycles at once.

3.3.3 Cache Analysis

An important part of the microarchitecture that needs attention during timing analysis are caches. The interface class `AbstractCache` models the abstract cache state of the cache under analysis and specifies the behaviour of the cache replacement policy.

- `update` models the effect of loading from or storing to a given abstract address, usually an address interval.
- `lessequal` and `join` specify the abstract analysis domain by providing basic lattice operations $\sqsubseteq$ and $\sqcup$.

For cache analysis with local classifications [54], the following functions are also relevant:

- `classify` tells for a given abstract address whether an access is guaranteed to hit or miss the cache.

For persistence cache analysis [62], the following functions are also relevant:

- `enterScope` and `leaveScope` model the behaviour on entering and leaving a persistence scope.
- `getPersistentScopes` determines for a given abstract address the scopes in which the address is persistent.

3.3.4 Path Analysis

To perform path analysis on the results of microarchitectural analysis, there are mostly two tasks to perform. The first is to determine a compact micorarchitectural state graph with what we call *weights* for each edge – collapsing multiple execution cycles at once. Weights can be as simple as numeric values such as the number of cycles, the number of cache misses, etc., but also more complex things such as the set of persistent cache misses. The second task is, to use these weights to build up the constraints of an integer linear program that encodes the worst-case path through the microarchitectural state graph. While solutions to the second task are rather specific to the constraints to be generated, solutions to the first task share sufficient commonalities to be captured by an interface class.

During construction of the microarchitectural state graph, we traverse the implicit graph at cycle granularity as described in Section 3.3.2. To keep the graph compact, LLVMTA joins nodes, i.e. microarchitectural states, and edges where possible (see `isJoinable`) without losing too much precision. The general interface class to construct the weights on these collapsed edges is `StateGraphEdgeWeightProvider`. There is a simpler but more limited interface class for numeric weights only, called `StateGraphNumericEdgeWeightProvider`.

- `extractWeight` takes a part of a microarchitectural state called *LocalMetrics* and extracts a weight (of any type) from it. The LocalMetrics can accumulate any weight within a state during microarchitectural analysis, e.g. the number of cycles executed or the number of cache misses since entering a basic block.
- `joinWeight` combines weights of to-be-collapsed edges with same source and same target node, e.g. by taking the maximum.

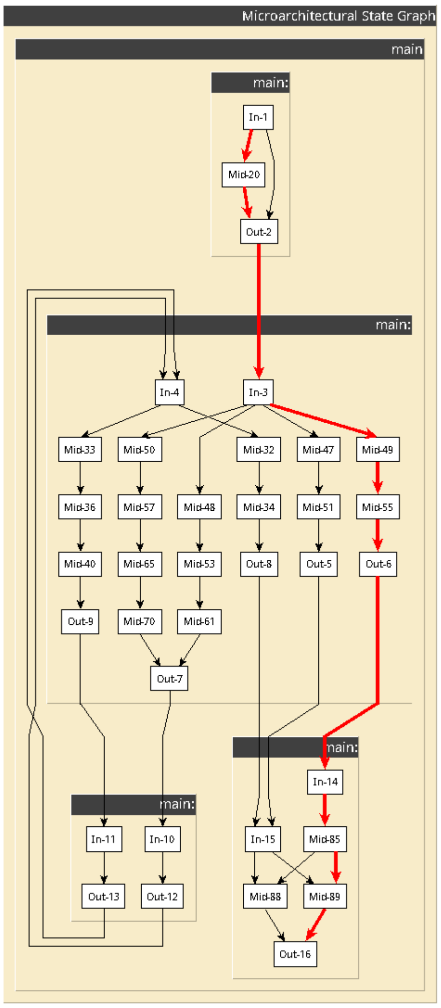

Figure 3 Simple AEG visualized with yComp.

- `concatWeight` concatenates two weights of two consecutive to-be-collapsed edges, e.g. by adding two numeric weights.
- `getNeutralWeight` returns the neutral element w.r.t. operation `concatWeight`.

4 Using LLVMTA

In this section, we shortly discuss the usage and the expected output of LLVMTA.

As input, a program written in C is provided to LLVMTA. CLANG translates the C program to its machine intermediate representation that is specific to the selected target architecture (ARM or RISC-V). The `runtestcase` script takes care of all necessary compilation steps and is the main script to perform timing analysis using LLVMTA. LLVMTA can automatically derive loop bounds using LLVM's scalar evolution analysis in many cases. If LLVMTA fails to obtain a loop bound, the user has to provide loop bounds manually in a CSV format `LoopAnnotations.csv`. The file can be auto generated with placeholders to fill the bounds using the `-ta-output-unknown-loops` option.

Table 1 Output files generated by LLVMTA.

Compiler Frontend	Assembler (`.txt`)	Program in LLVM's machine intermediate representation
	Assembler (`.S`)	Unlinked program assembly
Preanalysis	AnnotatedHeuristics	Contains inserted partitioning directives guiding context-sensitive analysis
	PersistenceScopes	Start and end of all persistence scopes
	CallGraph	List of all statically known callers and callees
	ConstantValueAnalysis	Constant values determined for machine registers and memory cells for each instruction
	LoopBounds	Contains loop bounds determined
	AddressInformation	Addresses of instructions and addresses of data accessed by memory operations
Microarch. Analysis	MicroArchAnalysis	Invariant set of microarchitectural state at the beginning and end of each basic block
	StateGraph_Time (`.vcg`)	Microarchitectural state graph
Path Analysis	LongestPath	ILP path analysis formulation
	PathAnalysis_<Weight> ..._<Max|Min>	Detailed results of the path analysis including worst-case path
Results	TotalBound (`.xml`)	Machine readable output of calculated bounds
	Statistics	Resource consumption of analysis

During the analysis execution, intermediate results are printed as the different analysis stages are performed. The output files are listed in Table 1. These files help the user understand what happens during the analysis and to inspect intermediate and final results. One important output is the file `StateGraph_Time`, which contains the abstract execution graph with microarchitectural states as nodes and edges with weights such as timing or number of cache misses. The graph can be viewed with the `yComp` (`https://pp.ipd.kit.edu/firm/yComp.html`) graph viewer [66], as shown in Figure 3 for the "simplewhile" example, which is provided along with the test suite of LLVMTA. The example consists of a single while loop, incrementing a single variable 50 times before terminating. The WCEP is highlighted by the red edges. It is worth noting that multiple abstract microarchitectural states are created for the while loop in the main function allowing for higher analysis precision than a simple "single execution time per basic block" path analysis scheme.

5 Existing Research Applications of LLVMTA

In this section, we briefly summarize research carried out with the help of LLVMTA.

Cache Analysis. Classically, *write-back caches* have not been used in hard real-time systems as it was not known how to model them in a sufficiently precise way during WCET analysis. This gap has been closed by combining two perspectives: a store-focussed one, which answers whether a store may dirtify a clean cache line, and an eviction-focussed one, which answers whether a cache miss may evict a dirty cache line and thus cause a write back [9].

We also employed LLVMTA in the development and evaluation of exact cache analysis, in particular of exact cache persistence analyses [73].

Compositional Analysis. LLVMTA supports compositional analysis approaches, i.e. several weights can be chosen for maximization such as useful cache blocks or accesses to the shared bus. The impact of such independent maximizations of different metrics on the efficiency and precision of WCET analysis has been explored in a master's thesis [21]. In addition, LLVMTA can sample *interference response curves* and calculate *compositional base bounds* based on the results of a microarchitectural analysis that safely covers all possible cases of temporal interference [34]. In this way, it bridges the gap between schedulability analyses, which typically rely on timing compositionality, and modern microarchitectures, which typically exhibit timing anomalies.

Calculation of Interference on Shared Resources. In the context of WCET analysis for multi-core processors, an important quantity is the amount of shared-resource interference that a concurrent processor core can generate while the program under WCET analysis is executed. To safely overapproximate worst-case interference generation scenarios, we generalized the well-known *implicit path enumeration technique* (IPET) [50] in a way that takes into account arbitrary subpaths of the abstract execution graph for the concurrent processor core [44]. As we assume that this generalized IPET does not scale to multiple real-world programs executed on a concurrent processor core, we sketched a *program-modular* calculation scheme [43] that calculates compositional base bounds per program on top of the generalized IPET.

Cache-Related Preemption Delay. In the presence of preemptive scheduling, preempting tasks evict cached memory blocks of preempted tasks, which have to be reloaded when the preempted tasks resume their execution [69]. This is commonly referred to as *cache-related preemption delay* (CRPD) [3]. We have experimentally evaluated the state-of-the-art techniques used to account for CRPD during timing analysis. Our experiments used task sets obtained by running LLVMTA on actual benchmarks. It turned out that the difference in precision of different CRPD analysis techniques and the overall impact of CRPD on schedulability are not as significant as observed for purely synthetically-generated task sets [69].

Strictly In-order Pipeline. To enable efficient and precise microarchitectural analysis, we introduced the strictly in-order pipeline in [35, 36]. This pipeline design enables an efficient progress-based abstraction and allows quantification of the effect of (amplifying) timing anomalies. The effect on WCET estimates and analysis performance have been evaluated using LLVMTA.

6 Conclusions

Since the kickoff of LLVMTA in 2014, it has been employed and extended in various research projects at Saarland University. It was exposed to TU Dortmund University in 2020. With the open source release of the code base in a version control system, and steady maintenance and integration, we foresee a great potential for LLVMTA to be used in research and education on WCET analysis more globally. Our short-term plan is to release practical exercises and tutorials for use of LLVMTA in education at the graduate level.

LLVMTA and a patched version of LLVM are available as open source for academic research purposes at:

https://gitlab.cs.uni-saarland.de/reineke/llvmta
https://gitlab.cs.uni-saarland.de/reineke/llvm

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of the Third International Static Analysis Symposium, SAS 1996, Aachen, Germany*, pages 52–66, 1996. doi:10.1007/3-540-61739-6_33.

2 Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: dynamic binary lifting and recompilation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 36:1–36:16. ACM, 2020. doi:10.1145/3342195.3387550.

3 Sebastian Altmeyer. *Analysis of preemptively scheduled hard real-time systems*. PhD thesis, Saarland University, 2013. URL: http://scidok.sulb.uni-saarland.de/volltexte/2013/5279/.

4 Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 295–308. ACM, 2013. doi:10.1145/2465351.2465380.

5 Mihail Asavoae, Claire Maiza, and Pascal Raymond. Program semantics in model-based WCET analysis: A state of the art perspective. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OASIcs*, pages 32–41. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/OASIcs.WCET.2013.32.

6 Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 1994, Oxford, UK*, pages 242–249, 1994. doi:10.1145/190347.190423.

7 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

8 Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011. doi:10.1007/978-3-642-18275-4_6.

9 Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in WCET analysis. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems, ECRTS 2017*, June 2017.

10 Silvian Calman and Jianwen Zhu. Interprocedural induction variable analysis based on interprocedural SSA form IR. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2010, Toronto, Ontario, Canada*, pages 37–44, 2010. doi:10.1145/1806672.1806680.

11 Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real Time Syst.*, 49(4):517–562, 2013. doi:10.1007/s11241-013-9178-0.

12 Antoine Colin and Isabelle Puaut. A modular & retargetable framework for tree-based WCET analysis. In *13th Euromicro Conference on Real-Time Systems (ECRTS 2001), 13-15 June 2001, Delft, The Netherlands, Proceedings*, pages 37–44. IEEE Computer Society, 2001. doi:10.1109/EMRTS.2001.933995.

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA*, pages 238–252, 1977. `doi:10.1145/512950.512973`.

14 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems*, 12(1s):40:1–40:25, 2013. `doi:10.1145/2435227.2435236`.

15 Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*, 2007. `doi:10.4230/OASIcs.WCET.2007.1193`.

16 Sun Ding, Hee Beng Kuan Tan, and Kaiping Liu. A survey of infeasible path detection. In Joaquim Filipe and Leszek A. Maciaszek, editors, *ENASE 2012 - Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering, Wroclaw, Poland, 29-30 June, 2012*, pages 43–52. SciTePress, 2012.

17 Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2002. URL: `http://nbn-resolving.de/urn:nbn:se:uu:diva-1832`.

18 Alexis Engelke, Dominik Okwieka, and Martin Schulz. Efficient LLVM-based dynamic binary translation. In Ben L. Titzer, Harry Xu, and Irene Zhang, editors, *VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021*, pages 165–171. ACM, 2021. `doi:10.1145/3453933.3454022`.

19 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*, 2007. `doi:10.4230/OASIcs.WCET.2007.1194`.

20 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real Time Syst.*, 46(2):251–300, 2010. `doi:10.1007/s11241-010-9101-x`.

21 Claus Faymonville. Evaluating compositional timing analyses. Master's thesis, Saarland University, 2015.

22 Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001. `doi:10.1007/3-540-45449-7_32`.

23 Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems, APLAS 2010, Shanghai, China*, pages 188–203, 2010. `doi:10.1007/978-3-642-17164-2_14`.

24 David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. Lossy compression for worst-case execution time analysis of PLRU caches. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versailles, France, October 8-10, 2014*, page 203. ACM, 2014. `doi:10.1145/2659787.2659807`.

25 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673, pages 120–136. Springer, 2009. `doi:10.1007/978-3-642-03237-0_10`.

26 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 155–164. IEEE Computer Society, 2010. `doi:10.1109/ECRTS.2010.8`.

27 Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 23–35. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.23.

28 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU cache: Challenging lru for predictability. *ACM Trans. Embed. Comput. Syst.*, 13(4s):123:1–123:26, April 2014. doi:10.1145/2584655.

29 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.073.

30 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS 2006, Rio de Janeiro, Brazil*, pages 57–66, 2006. doi:10.1109/RTSS.2006.12.

31 Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2003, Guadalajara, Mexico*, pages 106–112, 2003. doi:10.1109/WORDS.2003.1218072.

32 Sebastian Hahn. *On static execution-time analysis.* PhD thesis, Saarland University, Saarbrücken, Germany, 2019. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27440.

33 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy*, pages 102–111, 2012. doi:10.1109/ECRTS.2012.14.

34 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France*, pages 299–308, 2016. doi:10.1145/2997465.2997471.

35 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 469–481. IEEE Computer Society, 2018. doi:10.1109/RTSS.2018.00060.

36 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: a provably timing-predictable pipelined processor core. *Real Time Syst.*, 56(2):207–245, 2020. doi:10.1007/s11241-019-09341-z.

37 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

38 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, volume 57 of *OASIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.WCET.2017.8.

39 Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999. doi:10.1109/12.743411.

40 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. doi:10.1109/JPROC.2003.814618.

41 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

42 Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In Youtao Zhang and Prasad Kulkarni, editors, *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, pages 43–52. ACM, 2014. `doi:10.1145/2597809.2597817`.

43 Michael Jacobs. *Design and Implementation of WCET Analyses: Including a Case Study on Multi-Core Processors with Shared Buses*. PhD thesis, Saarland University, 2021. `doi:10.22028/D291-34893`.

44 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23nd International Conference on Real-Time Networks and Systems, RTNS 2015, Lille, France*, 2015.

45 Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2009, Savannah, GA, USA*, pages 214–228, 2009. `doi:10.1007/978-3-540-93900-9_19`.

46 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund M. Clarke, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011. `doi:10.1007/978-3-642-29709-0_20`.

47 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Second IEEE / ACM International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA*, pages 75–88, 2004. `doi:10.1109/CGO.2004.1281665`.

48 Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006. `doi:10.1007/s11241-006-9205-5`.

49 Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007. `doi:10.1016/j.scico.2007.01.014`.

50 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems, LCT-RTS 1995, La Jolla, California*, pages 88–98, 1995. `doi:10.1145/216636.216666`.

51 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. *IEEE Trans. Software Eng.*, 21(7):593–604, 1995. `doi:10.1109/32.392980`.

52 Björn Lisper. SWEET - A tool for WCET flow analysis (extended abstract). In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer, 2014. `doi:10.1007/978-3-662-45231-8_38`.

53 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS 1999, Phoenix, AZ, USA*, pages 12–21, 1999. `doi:10.1109/REAL.1999.818824`.

54 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems LITES*, 3(1):05:1–05:48, 2016. `doi:10.4230/LITES-v003-i001-a005`.

55 Florian Martin, Martin Helmut Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998. doi:10.1007/BFb0026424.

56 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK*, pages 5–20, 2005. doi:10.1007/978-3-540-31987-0_2.

57 Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. Technical report, Institut für Statistik und Wahrscheinlichkeitstheorie, Vienna University of Technology, 2012.

58 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008. doi:10.1109/RTCSA.2008.53.

59 Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne De Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in WCET analysis. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 3:1–3:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.3.

60 Peter P. Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013, Paderborn, Germany, June 19-21, 2013*, pages 1–8. IEEE Computer Society, 2013. doi:10.1109/ISORC.2013.6913220.

61 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. Improving WCET evaluation using linear relation analysis. *Leibniz Trans. Embed. Syst.*, 6(1):02:1–02:28, 2019. doi:10.4230/LITES-v006-i001-a002.

62 Jan Reineke. The semantic foundations and a landscape of cache-persistence analyses. *Leibniz Trans. Embed. Syst.*, 5(1):03:1–03:52, 2018. doi:10.4230/LITES-v005-i001-a003.

63 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007. doi:10.1007/s11241-007-9032-3.

64 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis, WCET 2006, Dresden, Germany*, 2006. doi:10.4230/OASIcs.WCET.2006.671.

65 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. *Trans. High Perform. Embed. Archit. Compil.*, 2:222–241, 2009. doi:10.1007/978-3-642-00904-4_12.

66 Georg Sander. Graph layout through the VCG tool. In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, pages 194–205, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

67 Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland*, pages 357–366. IEEE Computer Society, 2011. doi:10.1109/WCRE.2011.50.

68 Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 185–195. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461326.

69 Darshit Shah, Sebastian Hahn, and Jan Reineke. Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:11, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2018.7`.

70 Micha Sharir and Amir Pnueli. Two approaches of interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

71 Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA*, pages 395–404, 2010. `doi:10.1109/RTSS.2010.8`.

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

73 Gregory Stock, Sebastian Hahn, and Jan Reineke. Cache persistence analysis: Finally exact. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 481–494. IEEE, 2019. `doi:10.1109/RTSS46320.2019.00049`.

74 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 358–363. ACM, 2006. `doi:10.1145/1146909.1147002`.

75 Henrik Theiling. Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 23–30. IEEE Computer Society, 2000. `doi:10.1109/RTCSA.2000.896367`.

76 Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis - Reconstruction from Binary Executables and Usage in ILP-based Path Analysis*. PhD thesis, Saarland University, 2002.

77 Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 144–153. IEEE Computer Society, 1998. `doi:10.1109/REAL.1998.739739`.

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

79 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. `doi:10.1007/978-3-319-63390-9_2`.

80 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for LRU caches. *Proc. ACM Program. Lang.*, 3(POPL):54:1–54:29, January 2019. `doi:10.1145/3290367`.

81 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`.

82 Stephan Wilhelm. *Symbolic representations in WCET analysis*. PhD thesis, Saarland University, 2012. URL: `http://scidok.sulb.uni-saarland.de/volltexte/2012/4914/`.

DELOOP: Automatic Flow Facts Computation Using Dynamic Symbolic Execution

Hazem Abaza[1]
TU Dortmund, Germany

Zain Alabedin Haj Hammadeh
Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany

Daniel Lüdtke
Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany

Abstract

Constructing a complete control-flow graph (CGF) and computing upper bounds on loops of a computing system are essential to safely estimate the worst-case execution time (WCET) of real-time tasks. WCETs are required for verifying the timing requirements of a real-time computing system. Therefore, we propose an analysis using dynamic symbolic execution (DSE) that detects and computes upper bounds on the loops, and resolves indirect jumps. The proposed analysis constructs and initializes memory models, then it uses a satisfiability modulo theories (SMT) solver to symbolically execute the instructions. The analysis showed higher precision in bounding loops of the Mälardalen benchmarks comparing to SWEET and oRange. We integrated our analysis with the OTAWA toolbox for performing a WCET analysis. Then, we used the proposed analysis for estimating the WCET of functions in a use case inspired by an aerospace project.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Software and its engineering → Real-time systems software

Keywords and phrases Real-Time, WCET, Symbolic execution

Digital Object Identifier 10.4230/OASIcs.WCET.2022.3

Acknowledgements The authors thank Prof. Dr. Heiko Falk for his valuable feedback. Also, they thank Patrick Kenny for proof-reading the paper.

1 Introduction

Timing analyses aim to verify the timing constraints of a computing system. A timing analysis should start with computing a safe upper bound on the worst-case execution time (WCET) of each task (or sub-task in the case of directed acyclic graph (DAG) tasks) in the computing system. Then, a response-time analysis or a schedulability test should follow considering the scheduling policy and the deadline of each task. Estimates of the WCET of tasks can be obtained by using measurement, static or hybrid methods. The applications may be complex, therefore, the choice of the best method is not straightforward. However, only the static methods can cover all corner cases and can therefore provide safe upper bounds on the WCETs. Also, the development process is iterative, hence, setting up a static analysis would potentially save time and effort after applying changes compared to using measurements.

[1] This author's contribution has been conducted at the German Aerospace Center (DLR) while pursuing his Master's degree

A static WCET analysis has to provide an abstract model of the micro-architecture including, e.g., pipeline and caches, and facts on the program flow. Flow facts include program control-flow and upper bounds on loops. The Implicit Path Enumeration technique (IPET) computes the WCET as an objective function maximization in an integer linear programming (ILP) problem of the abstract interpretation of the micro-architecture and the execution paths of the program [19]. This paper presents an analysis based on dynamic symbolic execution (DSE) to automatically 1) compute upper bounds on loops and; 2) resolve indirect jumps to construct the control flow of the program. Automatic loop bounding and indirect jump resolution are desirable over manual annotation, which is error-prone and sometimes not manageable due to the amount of annotation needed [8].

DSE is a systematic approach to explore program paths and defining predicates [4]. A satisfiability modulo theories (SMT) [7] solver checks the satisfiability of the predicates to identify the next path. DSE has been used widely in computer security for, e.g., vulnerability discovery and reverse-engineering [27]. We use DSE in this work to explore program paths to identify potential jump targets and compute loop bounds. DSE reports results based on the given input values to the program, therefore, it cannot guarantee computing a safe upper bound on the loop bounds for applications implemented as an input-value-based state machine. In such applications, a value analysis should support DSE. However, applications that are implemented following the data-flow programming paradigm can use our DSE-based analysis safely as long as the control flow is input-value independent. In this work, we have special interest in data-flow applications, such as some on-board data processing (OBDP) applications. Hence, a value analysis is beyond the scope of this paper.

Developing embedded software using the inversion control programming principle improves modularity and maintainability [10]. Therefore, it is not uncommon nowadays to develop embedded software using e.g. C++-based software frameworks. C++-based software frameworks are the main motivation for this work. The German Aerospace Center (DLR) has developed a C++ software framework for developing OBDP applications, called Tasking Framework [17]. We will use it in this paper as a case study. Modularity and maintainability come at the cost of the underlying complexity. Therefore, performing static WCET analysis for such software is challenging. The challenges can be narrowed down to:

- **Control-flow reconstruction due to indirect jumps**
  Indirect jumps result mainly from virtual methods. They ensure that the correct function is called for an object. Calling a virtual method is translated at the binary level to an indirect jump instruction, in which the memory location of the target function is stored in a register. In Listing 1, the function *synchronizeStart()* in the Tasking Framework is defined as a virtual method. Listing 2 shows in Line 3 how the call is translated to an indirect jump in assembly. Such as branching instruction is challenging for the static analysis as it fails to fully construct the control-flow graph (CFG).

Listing 1 Indirect jump inside a simple for-loop where the bound is known at compile time.

```
void Tasking::TaskImpl::synchronizeStart(void){
for (unsigned int i = 0; (i < inputs.size()); i++){
        static_cast<ProtectedInputAccess&>(inputs[i]).synchronizeStart();}}
```

Listing 2 Indirect jump in the assembly code.

```
1 00009cca        ldr r3,[r3,0x7ff000000000]
2 00009ccc        move r0, r2
3 00009cce        blx r3
```

- **Loop Bounding**
  Loops that iterate over lists as shown in Listing 3 are specially challenging source-level loop bounding tools. The information about the list's size and its location in memory is not always available at the source level and requires additional binary level analysis to extract. Even simple *for* loops like the one presented in Listing 1 may be bounded by an object's value, which requires knowledge of the content of the memory location where the object is stored. Moreover, some loops are only available at the binary level. For example, constructing **n** objects from the same class sometimes is translated into loops at the binary level. These loops are hard to detect and bound at the source level.

Listing 3 A loop iterates over a bounded list.

```
//The loop iterates over the associated inputs to notify the task.
void Tasking::Channel::push(void) {
for (InputImpl* i = m_inputs; i != NULL; i = i->channelNextInput){
        i->notifyInput();}}
```

Our analysis uses a low level intermediate representation (LLIR) of the analyzed program as input. It translates each instruction into an SMT formula and symbolically executes them. We build a memory model, stack model, and register model to enhance the DSE such that each SMT formula updates the memory, stack and register models accordingly. With the help of a loop detection algorithm, namely Johnson's Algorithm [20], we bound loops.

We evaluated our analysis on the Mälardalen benchmark and compared the results with other tools, e.g., oRange [5]. The results showed high precision in bounding loops. We used the proposed analysis to provide flow facts to the open-source toolbox OTAWA [2]. Then OTAWA was used to compute the WCET of some Tasking Framework methods for the Cortex M3 architecture.

The rest of the paper is organized as follows: Chapter 2 visits the related work. In Chapter 3, we present our DSE-based analysis to compute loop bounds and resolve indirect jumps. The proposed analysis is evaluated in Chapter 4. Chapter 5 concludes the paper.

2 Related Work

In the scientific literature, SMT has been used to expose the program semantics to improve the tightness of the computed WCETs by eliminating infeasible paths. In [24], Ruiz et al. worked on machine code where they formulated the program states as sets of predicates to expose infeasible paths using SMT solvers. Henry et al. in [18] formulated the problem of computing the WCET as optimization modulo theory, which extends the satisfiability modulo theory. Neither paper addressed the problem of resolving indirect jumps. In [18], the loops must be unrolled before applying the proposed analysis. The analysis of program semantics is admitted to be easier at the source level [23]. However, for C++ software frameworks, performing the analysis at LLIR level is easier than at source level due to the complexity of the C++ language.

Gustafsson et al. presented in [16] an automated analysis to derive loop bounds using *abstract execution.* However, the proposed analysis was not developed to bound loops that iterate over a bounded list like in Listing 3. Therefore, we doubt that the polynomial correlations from the abstract execution can comprehend such loops. Besides that, the analysis was not developed to resolve potential indirect jumps in the CFG.

In many aerospace projects, intensive measurements are applied to estimate the WCET [12] using commercial tools like RapiTime [22]. Applying static analysis is done on critical functions [13]. Using aiT [11] is common to that end. Both approaches need human interaction, e.g., manual annotation. This work aims to automate the flow facts computation and to use the open-source toolbox OTAWA.

3 DSE-based Flow Fact Computation

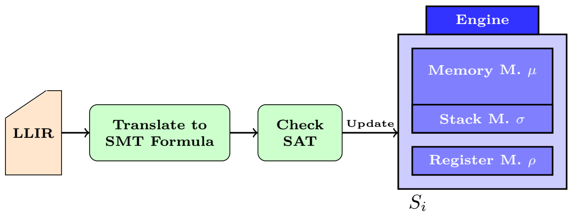

Figure 1 Analysis steps in DELOOP with the engine state.

In this section, we elaborate on our proposed analysis: Dynamic symbolic Execution-based LOOP bounding (DELOOP). The analysis steps are shown in Figure 1. DELOOP takes the executable binary of the given program as input, computes loop bounds and resolves indirect jumps. The analysis carries out the following steps:

1. Lifting the executable binary to *static single-assignment (SSA)* LLIR. We use the commercial tool BINARYNINJA [3] for that purpose. Performing the analysis on LLIR makes the analysis platform-independent.
2. Detecting the loops using Johnson's Algorithm.
3. Translating each SSA instruction in the LLIR into SMT formulas. We use Microsoft Z3 [6] as the SMT solver.
4. Building and initializing memory, stack and register models as arrays of bit vectors. The models will store the state of the memory, stack and registers.
5. Symbolically executing each instruction by checking the satisfiability of the equivalent SMT formula and updating the affected model.

After lifting the executable binary of the given program, the CFG is reconstructed. DELOOP computes an upper bound on the number of executions for each *basic block*. Combined with the loop detection algorithm, DELOOP can report an upper bound on loops. The lifting tool, BINARYNINJA, is a reverse engineering framework used mainly for binary analysis. We used its Python API to parse the assembly code and facilitate all parts of the analysis.

3.1 Loop Detection

We implemented Johnson's Algorithm to detect loops in the given CFG. The algorithm takes the CFG as a directed graph $\mathbf{G}$ $(\mathbf{V}, \mathbf{E})$, which consists of a non-empty set of vertices $\mathbf{V}$ and a set of ordered pairs of vertices called edges $\mathbf{E}$. The algorithm can detect the loops, known as *elementary circuits*, within a time bounded by $\mathbf{O}((\mathbf{n}+\mathbf{e})(\mathbf{c}+\mathbf{1}))$ and space by $\mathbf{O}(\mathbf{n}+\mathbf{e})$, where $\mathbf{n}$ is the number of vertices, $\mathbf{e}$ the number of edges and $\mathbf{c}$ the elementary circuits in the graph. A single elementary circuit is defined as a closed path where no node appears twice, except that the first and last nodes are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other.

DELOOP groups the basic blocks in a single elementary circuit (i.e., loop). Each detected loop, denoted by λ, is given a loop ID that is equal to theID of the last basic block in the loop. Recursive function calls are not handled with the loop detection algorithm. However, DELOOP can automatically bound the depth of recursion during the DSE phase.

3.2 SMT formulas and engine state

To symbolically execute the program, we compile the SSA LLIR into SMT formulae. The SSA form of the LLIR facilitates the whole translation process as every SSA instruction is directly mapped to one SMT formula using *array* and *bit vector* theories.

Two memory models are built based on the array theory. Data inside the arrays are formulated as bit vectors with a *size* that matches the target architecture; thus, the arrays are defined as arrays of bit vectors. The first memory is used for symbolic execution of the load/store instructions and is initialized with the values of all the program's data variables in the given executable binary. The second memory, the stack, is dedicated for the push/pop instructions. Both memory models grow and are updated dynamically along the DSE of the program.

Besides the models for memory and stack, we have a third model for representing the registers and flags. This model is also updated dynamically. Together, the memory model μ, the stack model σ and the register model ρ represent the *engine state* S. SSA instructions are translated to formulas in a form that implies the mathematical effect of the SSA instruction on the engine state. For example, the SSA instruction $R_2 = R_3 + 1$ is translated as shown in Equation 1 where bit vector variables are defined for R_2, R_3 and the immediate value.

$$R_2 = R_3 + 1 \implies BitVec(R_2, size) = BitVec(R_3, size) + BitVec(1, size) \tag{1}$$

Memory instructions are also interpreted in the same way. For example, the SSA instruction shown in Equation 2 is computed as *select(mem,*$0x8080$*)* where *mem* is the memory model and $0x8080$ is the load address. The translator performs the previous steps for all kinds of LLIR operations.

$$R_2 = [data\_0x8080] \implies BitVec(R_2, size) = select(mem, 0x8080) \tag{2}$$

3.3 Dynamic symbolic execution

DSE is used in a number of industrial tools to explore the CFG of a sequential program **P** for identifying test inputs that can lead the execution to new paths [7]. A path Π in the program **P** is said to be feasible if there is a non-empty set of inputs I such that $\forall i \in I$ the execution of **P** follows the path $\mathbf{\Pi}$. If $I = \emptyset$, then the path is not feasible.

Inspired by that concept, we try to explore loop bounds. For a program **P** starting at an initial path $\mathbf{\Pi_{in}}$ with a set of initial inputs I_{in}, we aim to deduce the set of outputs at the end of the path $\mathbf{\Pi_{in}}$: I_{out}. Our approach uses I_{out} as the new I_{in} to reach the next path. Following this concept, we dynamically execute all the feasible paths in the given CFG.

DELOOP checks the satisfiability of every SMT formula and updates the engine state S with the effect of execution. The SMT formulas are categorized into four main types: memory-related, stack-related, register-related and director formulas. Director formulas represent the branching instructions and are responsible for setting the execution path for the solver. Memory-related formulas update the memory model μ in the engine state. Similarly, stack and registers-related formulas update the stack σ and register ρ models respectively.

The concept of states transformed our execution from a static to a dynamic symbolic execution. For example, during the translation of $R_2 = R_3 + 1$, the translator first checks whether there are previous variables in the engine state for R_3 and R_2. In the case of already existing variables, the value of R_3 is fetched from ρ and increased by one and then assigned to R_2. If R_3 has a previous value of 100, then the translation process is done as follows:

$$R_2 = R_3 + 1 \implies BitVec(R_2, size) = BitVec(100, size) + BitVec(1, size) \tag{3}$$

The same is true for the memory instruction in Equation 2. If the address $0x8080$ has a value, let it be $0xa080$, then R_2 will be updated as follows: $R_2 = [data\_0x8080] \implies 0xa080$.

3.3.1 Bounding loops

The execution starts from the program entry point and continues to the CFG's exit function, or to the synthetically inserted exit point, which can be defined by the person who performs the analysis to stop the analysis at a designated point. DELOOP symbolically executes each SSA instruction and updates the engine state. Also, for each basic block B_i, DELOOP stores the number of executions EX_i of B_i. After finishing executing, the loops that are detected by Johnson's Algorithm, are visited and the bound is computed as the maximum number of executions for each basic block in loop λ. Let $\bar{\beta}$ be a function that returns an upper bound for a given loop λ:

$$\bar{\beta}(\lambda) = \max_{\forall B_i \in \lambda} \{EX_i\} \tag{4}$$

In the case of nested loops, Equation 4 returns the total number of executions of the inner loop, which is a non-necessary over-approximation. Therefore, before reporting the loop bounds we check if there are nested loops and update the loop bounds of inner loops as follows: $\bar{\beta}(\lambda_{inner}) = \bar{\beta}(\lambda_{inner})/\bar{\beta}(\lambda_{outer})$

3.3.2 Indirect jumps

Symbolic execution builds correlations between basic blocks for the program under analysis. It generates equations depending on an input variable to describe the jump target and the execution sequence of the program. These correlations can be used to resolve indirect jumps and anticipate the next basic block to be executed. However, the static symbolic execution generates multiple equations, based on the input and CFG path, that may satisfy the jump target resolution. These equations can be represented as first-degree-polynomial equations in the form of $a + x * C$ where a is the base of the jump table and $x * C$ is an offset. In each SMT formulated equation, C will depend on the input and the CFG path. The dynamic symbolic execution narrows the search space for these equations as it defines the execution path based on the given inputs for every solution iteration. In our generated engine model, the value of the indirect jump register is being updated based on the SAT formulations from state i till the indirect jump call instruction. That implicitly resolves the generated SAT inter-basic block formulations.

During the execution in our execution model, the indirect jump target is correlated to the CFG and the input through the forward propagation of the data. The result correlation is an SMT formulation of bit vectors and memory arrays. To resolve the formulation into meaningful targets, a reversed data-flow analysis with defined stop conditions needs to be run. However, this solution will lead to multiple resolutions for the formulation with no SAT guarantees. The dynamic symbolic solution solves this problem through the forward update of the engine states.

$$\begin{split} call(R3) \implies BitVec(R_3, size) = BitVec(select(mem, 0x8080), size) + \\ BitVec(select(mem, BitVec(R_1, size)), size) \end{split} \tag{5}$$

The update of the state after each execution implicitly preserves forward propagation of the memory arrays and bit vector values that will correctly resolve the jump target. For example, an indirect jump call formulation as in Equation 5 can be resolved to the jump target address by substituting the propagated values of the memory address and R_1 at the engine state executing the indirect call instruction.

Table 1 Benchmark results where L: loops; E: exact bounding.

Program	*#L*	*E*	*Program*	*#L*	*E*	*Program*	*#L*	*E*
adpcm	27	27	bs	1	1	cnt	4	4
cover	3	3	crc	6	6	duff	2	2
edn	12	12	expint	3	3	fac	1	1
fdct	2	2	fft1	30	30	fibcal	1	1
fir	2	2	insort	2	2	jcomplex	2	2
ludcmp	11	11	matmult	7	7	ndes	12	12
ns	4	4	nsichneu	1	0	prime	2	2
qsort-exam	6	6	qurt	3	3	select	4	4
ud	11	11						

Table 2 Loop-bounding tools comparison where BLT: bounded loop total.

Tool	BLT	% BLT	E	% E
DELOOP	158	99%	158	99%
oRange [5]	134	84%	117	73.5%
SWEET [9]	100	63%	81	51%

4 Evaluation

4.1 Mälardalen WCET benchmarks

The Mälardalen WCET benchmarks [15] are open-source test programs for WCET analysis. Although the Mälardalen WCET benchmarks are ANSI-C code, they can be used to verify our tool and compare its results against the state-of-the art tools. For validating our tool, we use Tasking Framework in the next section.

We used 25 programs from the Mälardalen WCET benchmark suite to test our tool. The results are presented in Table 1. E represents the number of loops which could be exactly bounded. For all programs except one, DELOOP can exactly bound the loops. For the very large function **nischneu**, the lifter, BINARYNINJA, failed to restore the CFG of the main function. It might not be surprising to exactly bound all the detected loops because we symbolically execute the program using the SMT formulas. In Table 2, we compare our results with oRange [5] and SWEET [9]. For oRange and SWEET, we recall the results from the cited papers. BLT and %BLT represent the number of bounded loops and percentage out of 159 loops respectively.

4.2 A use case developed using Tasking Framework

Tasking Framework [17] is an open-source [14] software development library. Also, it is a multithreading event-driven execution platform for embedded software. It provides abstract classes with virtual methods to realize an application by a directed graph of connected *tasks* and *channels*, where each computation block of a software component is realized by the class *task*, and the data exchanged between tasks is an object of the class *channel*. Periodic tasks are connected to a source of events as shown in Figure 3. Tasks can start executing as soon as their input data is available, thus, some of them can work concurrently. A task forwards the data to the next task by pushing it to the associated channel, which represents an interface between two tasks, and activating the next task. This data-driven activation mechanism is implemented in Tasking Framework with different activation semantics, e.g., and, or semantics.

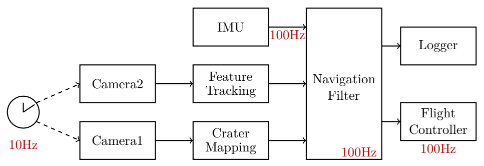

Figure 2 Use case inspired from the optical navigation sub-system in the ATON project [25].

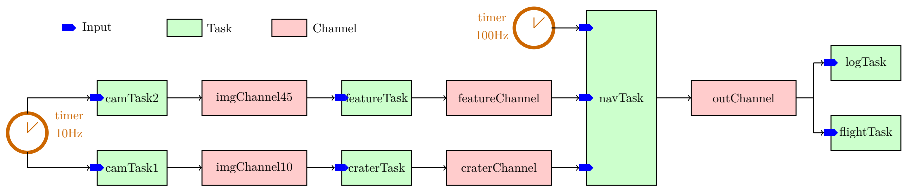

Figure 3 The use case in Figure 2 as realized by the Tasking Framework.

Tasking Framework has been used for many real-world aerospace applications such as Autonomous Terrain-based Optical Navigation (ATON)[25] and Scalable On-Board Computing for Space Avionics (ScOSA)[21]. ScOSA is an ongoing project in 2022.

We evaluated our analysis on a use case inspired from the optical navigation sub-system in the ATON project [25], and implemented using the Tasking Framework. In this sub-system, two camera drivers, *camTask1* and *camTask2*, run periodically and transfer the images to 1) a crater navigation component *craterTask* and 2) a feature tracking component *featureTask* respectively. The output of these components feeds the navigation filter *navTask* to estimate the position. The output is logged by *logTask* and forwarded to the flight controller *flightTask*.

4.2.1 Results

- **SWEET:** Its input is an IR based on the ARTIST2 Language for Flow Analysis (ALF). To apply SWEET, we built the binary code, then lifted it to LLVM using RetDec [1], which is a retargetable machine code decompiler based on LLVM. We translate the LLVM IR to ALF using the translator introduced in [26]. SWEET failed to build its abstract execution model.
- **oRange:** We generated the binary code and lifted it back to C code using RetDec. oRange reports *NOCOMP* for all loops in the use case.
- **DELOOP:** We integrated DELOOP with OTAWA as shown in Figure 4 to compute the WCET.

The results are presented here:

- **Loops:** Unlike the loops in the benchmark, Tasking Framework does not contain any simple loop like the one in Listing 4. The loops in Tasking Framework are either bounded by an object's attribute, see Listing 1, or iterates over a list, see Listing 3. However, the code of the user-developed tasks may contain different types of loops.

Listing 4 Simple ANSI-C loop.

```
for (int i=0; i<20 ; i++){}
```

DELOOP provides more than one bound for loops, one bound per instance. For example, each channel in our case study will run its own copy of the *push()* function; thus, the loop in Listing 1 will be executed by different tasks in the case study. DELOOP will compute an upper bound for each copy of the loop. The loop is bounded by the number of associated inputs and is thus bounded by *two* for the *navTask* while it is bounded by *one* for all other tasks.

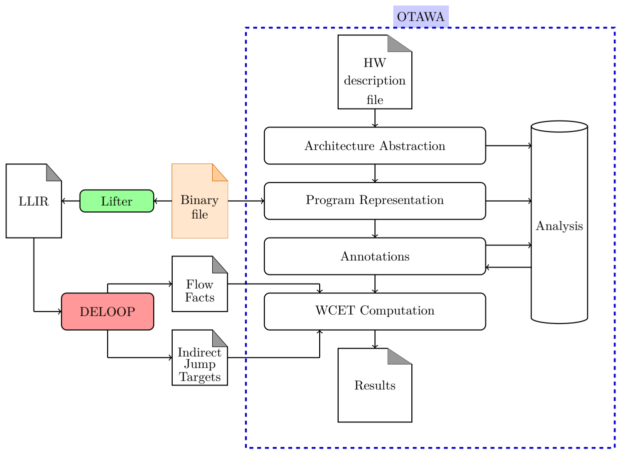

Figure 4 DELOOP integrated with OTAWA.

Also, DELOOP detected an implicit loop, which does not appear in the source code, as shown in Listing 5. *navTask* has *three* input objects, thus, the bound of this loop is *three*.

Listing 5 A constructor template translated into a loop in assembly code.

```
template<size_t n>
InputArrayProvider<n>::InputArrayProvider(void):
InputArray(inputMemory, n) {}
```

- **Indirect jumps:** The indirect jumps in Tasking Framework are mainly due to virtual methods. Virtual methods are there to support, for instance, three scheduling policies. After compilation, each indirect jump has only one target. Therefore, resolving the indirect jumps using DSE is safe. All the indirect jumps in our case study were resolved.
- **WCET Computation:** As mentioned earlier in this paper, we use OTAWA as a static analyzer and DELOOP as a flow facts generator as shown in Figure 4. This setup expands the capabilities of OTAWA in estimating WCET for C++ code. After given OTAWA a hardware description file for *armv-7m*, the WCET estimation starts with reconstructing the CFG. Then, the results of the loop analysis performed by DELOOP are passed to OTAWA for the WCET analysis. The analysis is performed for a bare-metal implementation.
  In OBDP applications based on a data-flow programming paradigm, ideally, each task pushes to the associated channel to activate the next task. This data-driven activation mechanism is implemented in Tasking Framework via the *push()* method. *push()* starts a chain of method calls, which ends with *queue()* that queues the next connected task in the ready queue. The chain contains two loops and one indirect jump. Bounding the WCET of *push()*, i.e., the chain of function calls, helps in estimating the overhead imposed by Tasking Framework. The implementation of *push()*[2] contains two loops: Loop1 is the outer loop that iterates over the tasks associated with the considered channel; Loop2 is executed for each iteration on Loop1 and it iterates over the inputs of each associated task with the considered channel. The WCET of *push()* executed by the task *camTask1* is 2435 cycles. Note that the channel *imgChannel10* is associated with only one input object, i.e. task *craterTask*. The same result is valid for the *push()* executed by the task

[2] https://github.com/DLR-SC/tasking-framework/commit/349ce3ddd98cd1fe69daf08318e1b8cbf9c01e9b

camTask2 because it has the same flow facts. The WCET of *push()* executed by the task *featureTask* and *craterTask* is 3635 cycles. Finally, the WCET of *push()* executed by the task *navTask* is 4800 cycles. Table 3 summarizes the results. As the results show, *push()* has different WCET values for different tasks, but it is bounded and fixed for each task.

Table 3 Results of the WCET analysis for the push function in the use case in Figure 3.

Task	Loop1	Loop2	WCET (cycles)
camTask1	1	1	2435
camTask2	1	1	2435
craterTask	1	3	3635
featureTask	1	3	3635
navTask	2	1	4800

- **Performance:** The analysis was executed on a workstation with Linux, i7-9750H processor and 16Gbyte RAM. The use case has a binary size = 664 kbyte. The analysis used 25% of the CPU capacity and 640 Mbyte of memory. The analysis took about 81 seconds to compute the flow facts.

5 Conclusions

The complexity of modern architectures, software development practices and compilers often leads to executable code which is difficult to match to its source code. Additionally, manual computation of flow facts and manual annotation are error-prone especially for software developed using object-oriented practices, in which one loop can be executed many times by different objects for different number of iterations. This provides motivation to compute the flow facts at the binary level.

In this work, we proposed an analysis to bounding loops and resolving indirect jumps using DSE. The proposed analysis lifts the executable binary to SSA LLIR, then each SSA instruction is translated into an SMT formula. Using the Z3 SMT solver, the satisfiability is checked and memory, stack and register custom models are updated accordingly. We showed that the proposed analysis can safely compute upper bounds on loops in the Mälardalen benchmarks. Also, we used the proposed analysis together with OTAWA to compute the WCETs for a use case developed using the Tasking Framework.

Although successful in computing loop bounds and resolving indirect jumps, the proposed analysis has two main limitations: 1) the need for value analysis for some applications to guarantee that the computed bounds are safe; 2) using a memory model, which might be very complex for large applications and therefore increase the analysis time. We will investigate in the future development the scalability of DELOOP to larger applications in our ScOSA project. Also, we are interested in verifying whether DELOOP yields any improvement in terms of WCET estimation by conducting more case studies for which oRange and SWEET can compute the flow facts.

References

1 Avast. RetDec. `https://github.com/avast/retdec`. [accessed May 03, 2022].

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

3 BINARYNINJA. Binary Ninja. `https://binary.ninja/`. [accessed May 03, 2022].

4 Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, 2016. `doi:10.1109/SANER.2016.43`.

5 Marianne de Michiel, Armelle Bonenfant, Hugues Casse, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166, 2008. `doi:10.1109/RTCSA.2008.53`.

6 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

7 Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. `doi:10.1145/1995376.1995394`.

8 Andreas Ermedahl and Jakob Engblom. Execution time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4:437–455, 2007.

9 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2007.1194`.

10 Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997. `doi:10.1145/262793.262798`.

11 Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static program analysis. In Renè Jacquart, editor, *Building the Information Society*, pages 377–383, Boston, MA, 2004. Springer US.

12 Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2012.81`.

13 Jorge Garrido, Juan Zamorano, and Juan A. de la Puente. Static analysis of WCET in a satellite software subsystem. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 87–96, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2013.87`.

14 German Aerospace Center (DLR). Tasking Framework. `https://github.com/DLR-SC/tasking-framework`. [accessed May 03, 2022].

15 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. `doi:10.4230/OASIcs.WCET.2010.136`.

16 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, 2006. `doi:10.1109/RTSS.2006.12`.

17 Zain Alabedin Haj Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, and Daniel Lüdtke. Event-driven multithreading execution platform for real-time on-board software systems. In *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-time Applications*, pages 29–34, 2019.

18 Julien Henry, Mihail Asavoae, David Monniaux, and Claire 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. `doi:10.1145/2666357.2597817`.

19 Hajer Herbegue, Hugues Cassé, Mamoun Filali, and Christine Rochange. Hardware architecture specification and constraint-based WCET computation. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 259–268. IEEE, 2013.

20 Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

21 Andreas Lund, Zain Alabedin Haj Hammadeh, Patrick Kenny, Vishav Vishav, Andrii Kovalov, Hannes Watolla, Andreas Gerndt, and Daniel Lüdtke. ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture. *CEAS Space Journal*, May 2021. `doi:10.1007/s12567-021-00371-7`.

22 RAPITASytems. RapiTime. `https://www.rapitasystems.com/products/rapitime`. [accessed May 03, 2022].

23 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. Improving WCET evaluation using linear relation analysis. *Leibniz Transactions on Embedded Systems*, 6(1):02:1–02:28, February 2019. `doi:10.4230/LITES-v006-i001-a002`.

24 Jordy Ruiz and Hugues Cassé. Using SMT solving for the lookup of infeasible paths in binary programs. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–104, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2015.95`.

25 Stephan Theil, N Ammann, Franz Andert, Tobias Franz, Hans Krüger, Hannah Lehner, Martin Lingenauber, Daniel Lüdtke, Bolko Maass, Carsten Paproth, et al. ATON (autonomous terrain-based optical navigation) for exploration missions: recent flight test results. *CEAS Space Journal*, 10(3):325–341, 2018.

26 Rick Veens. Adding support for static WCET analysis to LLVM, 2018. Master's thesis. URL: `https://research.tue.nl/en/studentTheses/adding-support-for-static-wcet-analysis-to-llvm`.

27 Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky, and Shamil Kurmangaleev. Sydr: Cutting edge dynamic symbolic execution. In *2020 Ivannikov Ispras Open Conference (ISPRAS)*, pages 46–54, 2020. `doi:10.1109/ISPRAS51486.2020.00014`.