

21st International Workshop on Worst-Case Execution Time Analysis

WCET 2023, July 11, 2023, Vienna, Austria

Edited by

Peter Wägemann



Editors

Peter Wägemann 

Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
waagemann@cs.fau.de

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 978-3-95977-293-8

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/978-3-95977-293-8>.

Publication date

July, 2023

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>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work
under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.WCET.2023.0

OASlcs – OpenAccess Series in Informatics

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

Editorial Board

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

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

■ Contents

Preface	
<i>Peter Wägemann</i>	0:vii
Committees	
.....	0:ix
 Regular Papers	
Warp-Level CFG Construction for GPU Kernel WCET Analysis	
<i>Louison Jeanmougin, Pascal Sotin, Christine Rochange, and Thomas Carle</i>	1:1–1:13
Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators	
<i>Alban Gruin, Thomas Carle, Christine Rochange, and Pascal Sainrat</i>	2:1–2:12
Exploring iGPU Memory Interference Response to L2 Cache Locking	
<i>Alfonso Mascareñas González, Jean-Baptiste Chaudron, Régine Leconte, Youcef Bouchebaba, and David Doose</i>	3:1–3:11
Clustering Solutions of Multiobjective Function Inlining Problem	
<i>Kateryna Muts and Heiko Falk</i>	4:1–4:12
Efficient and Effective Multi-Objective Optimization for Real-Time Multi-Task Systems	
<i>Shashank Jadhav and Heiko Falk</i>	5:1–5:12
Towards Multi-Objective Dynamic SPM Allocation	
<i>Shashank Jadhav and Heiko Falk</i>	6:1–6:12
Constant-Loop Dominators for Single-Path Code Optimization	
<i>Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner</i>	7:1–7:13
Analyzing the Stability of Relative Performance Differences Between Cloud and Embedded Environments	
<i>Rumen Rumenov Kolev and Christopher Helpa</i>	8:1–8:12
EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications	
<i>Simon Wegener, Kris K. Nikov, Jose Nunez-Yanez, and Kerstin Eder</i>	9:1–9:14



■ Preface

Welcome to the Proceedings of the 21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023). The 21st edition of the WCET Workshop is held on July 11, 2023, and is co-located with the Euromicro Conference on Real-Time Systems (ECRTS 2023) in Vienna, Austria. The WCET Workshop targets research on worst-case execution time analysis in the broad sense and serves as an annual meeting for the WCET community. In this year's edition, the scope of the workshop has been broadened, and the call for papers also welcomed contributions on analysis techniques for resources other than time, such as energy consumption.

The WCET Workshop has the goal of bringing together people from academia and industry. This goal is also reflected in the composition of the program committee (listed below) with members from academia, research institutes, and industry.

This year, the workshop received 12 submissions, each of which received at least three reviews. Based on these reviews and an online discussion, the program committee selected nine papers to appear for presentation at the workshop and in these proceedings. These papers cover a wide range of topics, including, among others, WCET analysis for GPU architectures, multi-objective optimization, and energy-consumption analysis.

Ensuring a high-quality program, organizing the WCET Workshop, and publishing open-access proceedings is a joint effort of many people: First, I would like to thank all members of the program committee and external reviewers for their time and effort in reviewing the submissions, providing comprehensive feedback, and participating in the online discussions. I am also very grateful for the support of the WCET Steering Committee throughout the organization of the workshop. Schloss Dagstuhl provided excellent assistance for the publishing process; many thanks to the whole team and especially to Michael Didas for the detailed and friendly support in preparing these proceedings. Finally, I especially thank all authors for contributing their work and you for your interest in these proceedings. I hope that these proceedings will be inspiring and helpful for your work in the future. It has been a pleasure for me to serve as a workshop chair for the WCET community.

Erlangen, Germany
June 23, 2023
Peter Wagemann



■ Committees

Program Chair

- Peter Wägemann, Friedrich-Alexander-Universität Erlangen-Nürnberg

Program Committee

- Konstantinos Bletsas, Polytechnic Institute of Porto (ISEP/IPP)
- Hugues Cassé, IRIT - Université de Toulouse
- Christian Dietrich, Technische Universität Hamburg
- Zain A. H. Hammadeh, German Aerospace Center (DLR)
- Sebastian Hahn, AbsInt Angewandte Informatik GmbH
- Björn Lisper, Mälardalen University
- Enrico Mezzetti, Barcelona Supercomputing Center
- Isabelle Puaut, University of Rennes/IRISA
- Peter Puschner, Vienna University of Technology
- Jan Reineke, Saarland University
- Christine Rochange, IRIT - Université de Toulouse
- Martin Schoeberl, Technical University of Denmark
- Peter Ulbrich, Technische Universität Dortmund

External Reviewers

- Eva Dengler
- Emad Maroun
- Phillip Raffeck

Steering Committee

- Björn Lisper, Mälardalen University
- Isabelle Puaut, University of Rennes/IRISA
- Jan Reineke, Saarland University



Warp-Level CFG Construction for GPU Kernel WCET Analysis

Louison Jeanmougin ✉

IRIT - Univ. Toulouse 3 - CNRS, France

Pascal Sotin ✉

IRIT - Univ. Toulouse 2 - CNRS, France

Christine Rochange ✉ 

IRIT - Univ. Toulouse 3 - CNRS, France

Thomas Carle ✉ 

IRIT - Univ. Toulouse 3 - CNRS, France

Abstract

We present an abstract interpretation technique to automatically build a Control Flow Graph (CFG) representation of the execution of a GPU kernel. GPUs implement an inherently parallel execution model, in which threads are grouped within so-called warps that execute in lockstep. This execution model enables the representation of the execution of the threads of a warp as a single CFG. However, thread divergence may appear within a warp and its effect must be captured explicitly within the CFG. Our method builds the CFG of a warp by applying abstract interpretation on the assembly (Nvidia SASS) code of a kernel, and by maintaining an abstract representation of which threads within the warp agree on which values. This allows the method to detect precisely the points in the program where thread divergence may occur, and avoid spurious reactivation edges in the CFG. We apply our technique on benchmark kernels as a proof-of-concept, and generate IPET systems using the resulting CFGs.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Theory of computation → Abstraction

Keywords and phrases Graphical Processing Unit (GPU), Control Flow Graphs (CFG), Worst-Case Execution Time (WCET), Program analysis

Digital Object Identifier 10.4230/OASICS.WCET.2023.1

Funding *Thomas Carle*: This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the MeSCAliNe (ANR-21-CE25-0012) project.

1 Introduction

The ever-growing need for computation power begs the question of adopting hardware accelerators in real-time embedded systems. In particular, Graphical Processing Units (GPUs) have gained traction as they combine massive parallelism and versatility. However, their adoption for safety-critical real-time systems requires the ability to derive safe Worst-Case Execution Time (WCET) bounds for the programs accelerated by GPUs. Traditional static WCET analysis targets the execution of a sequential thread running in isolation on a CPU core. In practice, the embedded program is modelled using a Control Flow Graph (CFG) that captures all the possible execution paths of the program in a condensed representation. Abstract interpretation techniques can be applied on this graph to determine properties of the program execution (e.g. loop bounds, infeasible paths, cache behavior). Each node of the CFG corresponds to a sequence of instructions of the program whose worst-case execution duration is derived using a model of the target hardware (in fully static methods) or using measurements (in hybrid methods). The CFG is ultimately used in the Implicit



© Louison Jeanmougin, Pascal Sotin, Christine Rochange, and Thomas Carle; licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 1; pp. 1:1–1:13

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Path Enumeration Technique (IPET) in order to generate an Integer Linear Program (ILP) system that captures the possible execution paths and combines them with the worst-case execution duration of the nodes of the CFG. The solution of this ILP system is the WCET bound for the considered program. Additional techniques can be used to account for the effect of concurrent threads running on the same core [1] or on different cores of a multi-core System-on-Chip [8, 11].

In order to handle thousands of threads in parallel, GPUs implement a complex execution model in which the threads executing a program are hierarchically subdivided into groups called thread blocks and warps. Thread blocks are dynamically distributed among the Streaming Multiprocessors (SM) composing a GPU following occupation rules defined in the GPU drivers [2, 12]. Within each block, the threads are also grouped into warps. Threads within a warp execute in lockstep: at each execution cycle, each multiprocessor elects a warp for execution, and all the threads within the elected warp execute the same instruction. This execution model is known as Single Instruction Multiple Threads (SIMT). Since all the threads within a warp execute the same instruction at the same time, it seems natural to derive the worst-case execution time of a warp in isolation, using a single CFG, and following the classical WCET analysis workflow. From this information, additional analyses can then be developed and applied to combine multiple warps running on the same GPU and derive a WCET for the complete application.

However, the SIMT execution model is subject to a phenomenon called thread divergence that impacts the control flow, and ultimately the execution time, by serializing the execution of the different branches of conditional branch statements when the threads within a warp do not agree on the value of the condition. For each thread taken separately this has no impact on the control flow, but at warp level, this serialization mechanism creates additional transitions in the control flow that must be accounted for in the warp-level CFG.

In this paper, we present an abstract interpretation technique that builds an accurate warp-level CFG directly from the assembly code (Nvidia SASS) of the application. This technique models the semantics of the SIMT execution model, and is able to determine a subset of the conditional branch instructions for which the threads are statically guaranteed to agree on the execution condition.

Building the CFG by abstract execution of the machine code in a CPU context is known to bring the following benefits [3]:

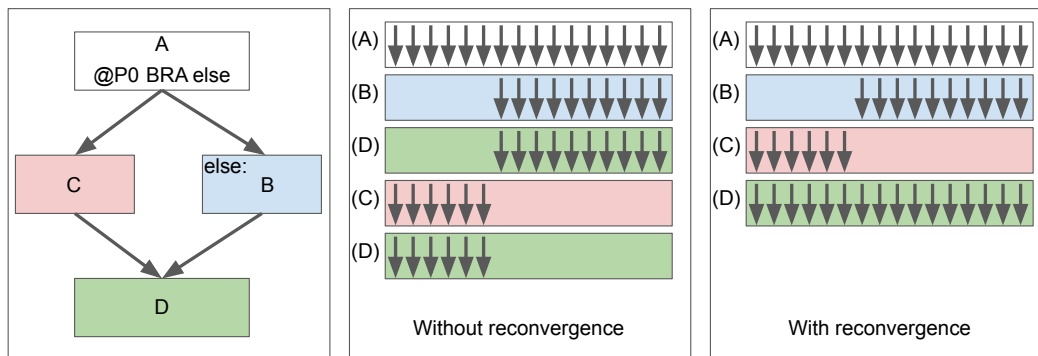
1. independence from the source language and the compilation process;
2. more accurate control graph and abstract values thanks to interleaving of value analysis and graph construction.

In the context of GPU timing analysis, this approach seems even more natural because:

- the source code describes the behavior of *each* thread, while the machine code describes the behavior of a warp;
- the subsequent WCET analysis is performed on the machine code¹

The paper is organized as follows. In Section 2, we present the details of the SIMT semantics. We then present our CFG construction method in Section 3 and evaluate it in Section 4. We present the related work on the topic of static WCET analysis of GPU kernels in Section 5 and we conclude in Section 6.

¹ We leave aside the problem of transferring the loop bound information from source to assembly code.



■ **Figure 1** Thread divergence example.

2 SIMT execution semantics

As mentioned earlier, GPUs implement a particular execution model called SIMT. At the highest level, the main CPU program calls a GPU function (called a kernel) that is executed by a specified number of threads. These threads are organized into blocks that are dynamically dispatched to the SMs composing the GPU. Within a block, threads are divided in groups of 32 (or 16 depending on the GPU architecture) called warps. Inside an SM, warp schedulers are responsible for selecting a warp to execute at each execution cycle. Threads within a warp execute in lockstep: whenever a warp is elected for execution, all the threads that compose it execute the same instruction. This greatly simplifies the logic, as all threads within a warp can be seen as sharing their program counter (PC). However this execution model can be problematic when the threads execute conditional branches: in certain situations, called thread divergence, threads within a warp may not agree on the value of the execution condition of a branch, and thus on the next value of their shared PC. This is handled by executing the branch with the threads that find the execution condition true, while masking the others, and then executing the fallback code with the other threads only. This mechanism is illustrated in Figure 1. On the left, the figure displays a simple CFG with a conditional branch: at the end of block A, the control flows towards B or C, and then reaches D regardless. At the end of block D, each thread executes the `EXIT` instruction that signals the end of execution for the thread. A possible execution for a warp is given in the middle. At the beginning, all threads within the warp² execute block A. Then the ten rightmost threads execute block B followed by block D, until they reach the end of the program. When the execution is over for these threads, the six leftmost threads execute block C, followed by block D. At this point, all threads have finished executing the kernel.

This serialization allows the correct execution of a kernel, but is not efficient, as some parts of the code (located after the if-then-else) are executed multiple times in sequence (e.g. block D in our example). A reconvergence mechanism is implemented to reduce the subsequent loss of performance: the compiler automatically detects areas in the code where thread divergence may occur (around conditional branch instructions) and adds special purpose instructions in the assembly code to re-synchronize the threads that have diverged. This is illustrated in the right part of Figure 1. As before, at the beginning all threads

² For space reasons we only depict 16 threads within the warp.

execute block A. Then the ten rightmost threads execute block B and are suspended, as they reach a reconvergence instruction at the end of the block. The six leftmost threads execute block C, and when they are done, all threads reconverge before executing block D.

The SASS instruction set (up to the Maxwell/Pascal ISA at least) contains two pairs of such instructions: **SSY/SYNC** and **PBK/BRK**. In each pair, the first instruction is used to signal a potential incoming divergence to the hardware, while the second instruction is used to synchronize diverging threads, thus forcing their reconvergence. In order to support nested conditional branches, an activation stack stores the necessary information: each entry in the stack is composed of a mask representing the active threads of the entry, and of the next PC value for these threads. The entry at the top of the stack always represents the currently active threads and the PC of the next instruction to execute. Additionally, each entry of the stack is typed in order to handle intertwined loop, if-then-else and break constructs:

- an entry is of **NIL** type if it does not correspond to a reconvergence point. The top of the stack is always of **NIL** type.
- an entry is of type **SYNC** if it was inserted using the **SSY** instruction. It usually is used to reconverge after a conditional branch due to a if-then-else or a loop construct.
- an entry is of type **BRK** if it was inserted using the **PBK** instruction. It usually is used to reconverge after a conditional break statement.

The **SSY** and **PBK** instructions contain the address of the instruction at which the corresponding reconvergence must occur.

The stack is maintained using the following rules:

- when a warp is mapped to an SM, a stack is allocated. It is initially composed of a single **NIL** entry with all threads active in the mask and the next PC corresponding to the start of the kernel code.
- when a warp reaches a **SSY @reconv_addr** (resp. **PBK @reconv_addr**) instruction, the next PC of the top entry becomes **reconv_addr**, and the entry becomes typed as **SYNC** (resp. **BRK**). This entry will be used when the threads reconverge. A new **NIL** entry is then pushed on the stack. This entry has the same thread mask as the previous top entry (i.e. all currently active threads remain active), and its next PC is set to the current PC + 8 (i.e. the next instruction in memory for 64 bit instructions). The **SSY** (resp. **PBK**) instruction prepares the stack for a potential divergence due to a future conditional branch instruction, but is not by itself a source of divergence.
- when a warp reaches a divergent conditional branch **BRA @addr**, the actual divergence must be accounted for in the stack. The top entry next PC is set to its current value + 8 (i.e. the address of the fallback code), and its mask is updated to contain only the active threads that do not take the branch. This entry will be used later to execute the fallback code. A new **NIL** entry is pushed to the stack. Its next PC is set to **addr** and its mask is composed of the active threads that take the branch.
- when a warp reaches a **SYNC** (resp. **BRK**) instruction, the active threads are removed from the mask of all entries in the stack, from the top and until a **SYNC** (resp. **BRK**) entry is reached. Each time a mask is modified, its corresponding entry is popped from the stack if the modified mask no longer contains any thread. In practice, the **SYNC** (resp. **BRK**) instruction suspends the execution of the currently active threads until a reconvergence point is reached by all the threads that must reconverge.
- when a warp reaches an **EXIT** instruction (i.e. the active threads reach the end of the kernel), all active warps that execute the **EXIT** are removed from the mask of all the entries in the stack. Once again, if a mask becomes empty doing so, its entry is removed from the stack.
- whenever an entry becomes the top entry, its type becomes **NIL** regardless of what it was before.

This set of rules is implemented in the hardware, and as the `SSY/SYNC`, `PBK/BRK` and `EXIT` instructions are automatically inserted by the compiler, the process of handling thread divergence when it occurs is transparent to the programmer. On the other hand, our static analyses follow closely this execution model in order to accurately account for its effect. In particular, to the best of our knowledge, it is the first time that a static analysis is performed at the granularity level of the assembly language for a GPU kernel and that the divergence/reconvergence instructions are taken into account. This is particularly important as in our experiments, we have encountered situations in which the compiler does not insert reconvergence instructions on the first post-dominating node after a conditional branch.

In the next section, we present our abstract interpretation method to build a warp-level CFG from the SASS code of a GPU kernel.

3 Abstract interpretation

In the manner of Reps and al. [3], we perform the CFG construction at the machine code level and we interleave exploration of the control flow with value analysis.

We construct the graph by starting at the entry point of the program with an initial abstract state representing the possible initial concrete states. We execute the instructions of the program on the abstract states and discover successors states. We do so until a fixpoint is reached and no more new states are discovered. An abstract state contains information on:

- the current program counter;
- the pending activation stack (see Section 3.1);
- several remarkable groups of threads (see Section 3.2);
- for each group, the registers on which these threads agree (see Section 3.3).

After presenting the base domains in the following subsections, we detail our abstract states in Section 3.4. For each domain, we provide a formal description of the concretization function γ , that associates a value in an abstract domain D^\sharp to a set of concrete values, and a description of its essential operations.

The notation $A \rightarrow B$ denotes a total function from A to B ; $A \dashrightarrow B$ denotes a partial function from A to B ; $\mathcal{P}(A)$ denotes a subset of A .

3.1 Activation stack abstract domain

We represent the possible configurations of the activation stack by a graph in which the nodes are composed of a control point and an optional tag (`SYNC`, `BRK` or `none`), plus two special nodes : `top` and `bot`. In such a graph, each path from `top` to `bot` represents the contents of a stack in a possible configuration. The stack concretization function γ_{Stack} takes as input a graph and returns the set of corresponding stack configurations.

$$\gamma_{\text{Stack}} : D_{\text{Stack}}^\sharp \rightarrow \mathcal{P}(\text{List}(\text{Act}^\sharp)) \quad \text{with } \text{Act}^\sharp \stackrel{\text{def}}{=} PC \times \text{Tag}$$

Operations

Most operations in this abstract domain are simple adaptations of classical operations on stacks. We detail the pop and filter operations.

Popping

While pushing an element on an abstract stack simply gives an abstract stack, popping the topmost element may yield several possible popped values and leave distinct remainders. It might also happen that the stack can be empty; *none* is then part of the result.

$$\text{pop}^\sharp : D_{\text{Stack}}^\sharp \rightarrow \mathcal{P}(\text{Act}^\sharp \times D_{\text{Stack}}^\sharp)$$

This operation is called after the active threads have been halted by an EXIT, SYNC or BRK. The execution may continue at any of the control point that can be popped. If the stack may be empty, this indicates that the program can halt here.

Filtering

When an operation halts the active threads, it removes them from the active mask, but also from all the masks in the stack, up to the bottom in case of an EXIT and up to the corresponding tag in case of a SYNC or BRK.

Our representations of the stack do not embed the masks³. However, we need to take into account that stages of the activation stack might have been removed after that their activation mask attained the zero vector.

We thus equipped our abstract stack domain with a filtering operation that processes the stages from the top of the stack up to an optional tag and for each stage take into account that its mask can/cannot/must reach zero. The latter information is taken as a parameter and is provided by the thread group abstract domain (Sec. 3.2).

$$\text{filter}^\sharp : D_{\text{Stack}}^\sharp \times \text{Tag} \times (\text{Act}^\sharp \rightarrow \{\text{keep, drop, any}\}) \rightarrow D_{\text{Stack}}^\sharp$$

3.2 Thread group abstract domain

In order to conduct a precise analysis, we sometimes need to retain the relations between certain groups⁴ of threads. We thus introduced an abstract domain that is able to remember relations like $A \subseteq B$, $A = B$ or $A \cap B = \emptyset$.

The domain does not keep track of the concrete threads present in a group, just the relations between these groups. The group names are stored in the abstract value. In our case, we use the special group `active` to denote the threads currently executing and one group per activation node in the stack graph to denote either its associated mask or an upper bound on it when in a cycle. This set of groups is denoted by G .

$$\gamma_{\text{Group}} : D_{\text{Group}}^\sharp \rightarrow \mathcal{P}(G \rightarrow \mathcal{P}(\text{Threads}))$$

We implement this domain by storing all the intersections of groups or complements of groups that must be empty. The constraint $A \cap B = \emptyset$ is stored as is, the constraint $A \subseteq B$ is stored as $A \cap \bar{B} = \emptyset$ and the constraint $A = B$ is stored as $A \subseteq B \wedge B \subseteq A$. The conjunction of constraints is an exact operation⁵ in this domain.

³ Putting the masks or abstraction thereof in the abstract stack would lead to very large graphs.

⁴ In this specific context, we use the term *group*, but it can be read as *set*.

⁵ An *exact operation* is an operation of the abstract domain that does not result in an over-approximation.

3.3 Agreement abstract domain

Thread divergence can occur when a conditional branch instruction is executed. However, in many situations all the threads of the warp agree on the predicate, by program design.

In order to determine if a divergence may occur or not, we created a new abstract domain that keeps track for a given group of threads of the registers on which these threads agree. We just store the name of these registers, not the values they contain. The beauty of this analysis is that we do not need to know the precise behaviour of each instruction but only what it reads, what it writes, and be sure that it is deterministic.

For example, if the instruction is `IADD R2, R4, R2`; we can tell that if a group of threads agree on `R2` and `R4`, they will all write the same value in `R2` and thus keep agreeing on these registers.

The basic version of this domain captures the identical registers in a given group of threads.

$$\begin{aligned} \gamma_{\text{Agree}} &: (\mathcal{P}(\text{Thread}) \times \mathcal{P}(\text{Reg})) \rightarrow \mathcal{P}(\text{Mem}) \\ \gamma_{\text{Agree}}(T, R) &= \{m \in \text{Mem} \mid \forall t_1, t_2 \in T, \forall r \in R, \text{read}(m, t_1, r) = \text{read}(m, t_2, r)\} \end{aligned}$$

Such abstract value is concretized as the set of memories such that any two threads in the group that read the same register on which the agreement was established, read the same value. Values in *Mem* describe both the registers and the DRAM memories of the GPU.

This domain is then lifted to handle several groups of threads, identified in Section 3.2.

$$\begin{aligned} D_{\text{GrAgr}}^\# &\stackrel{\text{def}}{=} G \rightarrow \mathcal{P}(\text{Reg}) \\ \gamma_{\text{GrAgr}} &: D_{\text{GrAgr}}^\# \rightarrow \mathcal{P}((G \rightarrow \mathcal{P}(\text{Threads})) \times \text{Mem}) \\ \gamma_{\text{GrAgr}}(f) &\stackrel{\text{def}}{=} \{ \langle g, m \rangle \in (G \rightarrow \mathcal{P}(\text{Threads})) \times \text{Mem} \mid \\ &\quad \text{dom}(g) = \text{dom}(f) \quad \wedge \quad \forall x \in \text{dom}(f), m \in \gamma_{\text{Agree}}(g(x), f(x)) \} \end{aligned}$$

When we process an instruction computing data, we update the information tied to each group in the following manner:

- If the group is equal to `active`, we perform a strong update. If the threads agree on the arguments, they gain or preserve the agreement on the result. If the threads disagree on at least one argument, agreement on the result is lost.
- If the group is not equal to `active` but intersecting it, we perform a weak update. It means that only a part of the group performs the instruction. Agreement on the result is thus lost.
- If the group is disjoint from `active`, we do not modify its agreements.

3.4 Warp state abstract domain

As announced on page 5, a state is made of the current program counter, the pending activation stack, a set of remarkable groups of threads and agreement information on these groups.

$$\begin{aligned} D_{\text{Warp}}^\# &\stackrel{\text{def}}{=} PC \times D_{\text{Stack}}^\# \times D_{\text{Group}}^\# \times D_{\text{GrAgr}}^\# \\ \gamma_{\text{Warp}} &: D_{\text{Warp}}^\# \rightarrow \mathcal{P}(\text{List}((PC \times \text{Tag}) \times \mathcal{P}(\text{Thread}))) \times \text{Mem} \\ \gamma_{\text{Warp}}(p, s^\#, g^\#, a^\#) &\stackrel{\text{def}}{=} \{ \langle \langle p, \text{none} \rangle, g(\text{active}) \rangle . s', m \mid \\ &\quad \langle g, m \rangle \in \gamma_{\text{GrAgr}}(a^\#) \quad \wedge \quad g \in \gamma_{\text{Group}}(g^\#) \\ &\quad \wedge \exists s \in \gamma_{\text{Stack}}(s^\#), \forall i, s'[i] = \langle s[i], g(s[i]) \rangle \} \end{aligned}$$

Processing an instruction

We define the abstract treatment of an instruction by separating the concerns of processing an instruction unconditionally (memo + operands) from the treatment of the condition. This organization of the abstract semantics avoids to consider for each kind of instruction a tedious and error-prone study of the interference between the instruction and the presence of a condition.

Managing the condition

Processing an instruction with its condition may produce several abstract values, that we do not wish to join immediately into a single one. An informal algorithm is given in Algo. 1 and it uses the **produce** keyword to signal one or several results (as **yield** in Python).

■ **Algorithm 1** Successors of an abstract state: condition management.

Input : An initial abstract state st_{ini}
Output : The production of one or more successor state

```

1  $PC_{cur} \leftarrow PC(st_{ini});$ 
2  $i \leftarrow$  instruction at  $PC_{cur};$ 
3  $st_{all} \leftarrow$  process unconditionally instruction  $i$  in state  $st_{ini};$ 
4 if  $st_{all}$  has no more active threads then
5   | produce all pop from  $st_{all};$ 
6 else
7   | produce  $st_{all};$ 
8 if  $i$  is conditional then
9   | produce  $st_{ini}$  with  $PC = PC_{cur} + 8;$ 
10  | if active threads might disagree on  $cond(i)$  in  $st_{ini}$  then
11  |   |  $st_{part} \leftarrow st_{ini}$  with a group skip separated from active;
12  |   |  $st_{some} \leftarrow$  process unconditionally instruction  $i$  in state  $st_{part};$ 
13  |   | if  $st_{some}$  has no more active threads then
14  |   |   | produce  $st_{some}$  with group skip renamed as active;
15  |   |   | else if  $PC(st_{some}) = PC_{cur} + 8$  then
16  |   |   |   | produce  $st_{some}$  with group skip folded into active;
17  |   |   |   | else
18  |   |   |   |   | produce  $st_{some}$  with group skip pushed as  $\langle PC_{cur} + 8, none \rangle;$ 

```

Processing unconditional instructions

1. Branching (BRA) replaces the current PC with the target of the instruction. Non-branching instructions increment the PC *in addition to their effect*.
2. Data-processing instructions (eg. IADD or LDC) modify the agreement component of the abstract value, as presented at the end of Section 3.3.
3. Reconvergence preparation instructions (SSY and PBK) push their target on the pending activation stack with a tag corresponding to the kind of synchronization.
4. Reconvergence instructions (SYNC and BRK) split the abstract stack on the first occurrence of the corresponding tag. The upper part is then filtered from the activation stages that can or must have their mask reduced to zero when we halt the threads in **active**.
5. The EXIT instruction filters the **active** threads from the whole stack.

In the next section we describe the results that we obtained using our method.

4 Evaluation

4.1 Experimentation on the Rodinia Benchmark

To the best of our knowledge, no GPU kernel benchmark dedicated to embedded or real-time systems has been released yet. To evaluate our analysis method, we thus used kernels extracted from the Rodinia [7] benchmark. Since our analysis does not support calls to kernels originated from another kernel (we are currently working on understanding the relationship between the call stack and the reconvergence stack), our evaluation was performed on 37 out of the 57 kernels composing Rodinia. For each of them, our prototype was able to generate a CFG, and from each CFG we generated an ILP system following the IPET method. Developing a loop-bound analysis for warp-level CFGs is part of future work, so for now we manually provided the loop bounds, and arbitrarily set each loop bound to 10 iterations (or a power of ten for nested loops). Loop reconstruction was done using [6]. Additionally, we arbitrarily set each instruction duration to 1 cycle in order to obtain durations for the blocks of the produced CFGs. Our objective was not to derive a real WCET for these kernels but to prove the feasibility of deriving a warp-level CFG and to use it in a standard IPET workflow.

Overall, most of the analyzed kernels have a very limited number of arcs modelling a possible divergence (34 of them have 4 or less of these arcs), which is coherent since most of these kernels are pretty simple, and do not feature if-then-else constructs. Interestingly, the 3 kernels with a slightly more complex control flow (switch-case and if-then-else constructs) have 39, 19 and 12 of them (respectively in the `mummergepuKernel`, `printKernel` and `reduce` kernels). This means that the kind of analysis that we propose is necessary in order to support even relatively simple kernels. For each of them we were able to compute a WCET, which shows that the generated CFGs are compatible with the IPET method. The CFG reconstruction took up to 1.3 seconds; in 75% of the benches, it took less than 90 ms.

In the future, we plan on looking at other benchmarks to try to find more complex kernels, or to develop our own benchmark with kernels that can be interesting for analysis or representative of embedded GPU kernels.

4.2 Evaluation of the abstract domain

In this subsection we give some results on the relative importance of the component of the abstract domain $D_{\text{Warp}}^{\#}$ presented in Section 3. We deactivate some parts of the abstract value and observe the fraction of the benchmark programs of Section 4.1 that see their WCET severely degraded. The results are the following:

Domain modification	WCET est. $\times 10$ or more
Limitation to acyclic graphs (Sec. 3.1)	25%
No group fine analysis (Sec. 3.2)	15%
No agreement analysis (Sec. 3.3)	52%
Agreement tracking only for <code>active</code>	37%

The results show that the agreement analysis is the key ingredient for the WCET analysis precision. We can also see that this analysis needs to be done not only for the active threads but also the pending threads. Eventually, we can spot that a precise group analysis is less crucial but should not be neglected.

5 State of the art

The problem of deriving a safe WCET bound for GPU kernels has so far been the topic of only a limited number of publications.

In [4], the authors tackle the problem by providing an ILP formulation that captures how a work-conserving warp scheduler could handle the workload corresponding to a kernel. The focus is put on how the scheduler hides the long latency instructions (e.g. memory accesses). However, in this preliminary work, the simplifying assumptions are very strong. In particular, the considered kernels are single path, which greatly simplifies the analysis and completely puts aside the problematic of thread divergence. The method proposed in [10] is also based on the modelling of the warp scheduler policy using an ILP system, focusing on the Greedy Then Round Robin scheduling policy. The authors propose analyses of the kernel code to handle memory access coalescing and roughly account for thread divergence. However, nothing is said about how thread divergence is detected, so we can assume that additional analyses such as the one that we propose are required.

In [5], a hybrid analysis method is used: a CFG is built to represent the execution paths of the kernel, and the execution duration of the basic blocks of the CFG are obtained using measurements. The authors propose an algorithm that extends the CFG of a single thread to a CFG that over-estimates the control transitions at warp-level, by adding extra edges that model thread divergence. This method is based solely on the topological properties of the thread CFG (i.e. divergence in the graph and dominance/post-dominance properties), and does not take into account the actual SIMT semantics, nor the fact that thread reconvergence only happens if and when SYNC/BRK instructions are inserted by the compiler. In our experience, the compiler does not always insert reconvergence instructions at the first post-dominant point in the CFG after a separation of paths in the graph, so the assumptions made in [5] may sometimes be too optimistic and lead to underestimations of the WCET. Moreover, since the authors do not follow closely the SIMT semantics, their method may add extra edges that are not added with our method a) when we detect an agreement between the threads of a warp and b) because when a conditional branch occurs, their algorithm does not know which branch is taken first.

An ad-hoc WCET analysis method for GPU kernels has been proposed in [9]. The algorithm builds on Single Static Assignment (SSA)-like analysis methods and on symbolical execution to statically detect the points in the CFG of a thread where agreement between all threads of a warp is statically guaranteed. This is, to the best of our knowledge, the only method (before ours) that tries to determine agreement on a condition when a conditional branch occurs. In comparison, our method determines agreement points by introducing the SIMT stack mechanism in our model, which provides more precise results: we determine agreement between active threads at each point, while the method based on SSA-like properties only allows to reason about all the threads in the warp. Moreover, our method aims at building a warp-level CFG that can then be used in a standard WCET analysis pipeline, so it can benefit from classical analyses on CFGs (e.g. cache analysis) and from the IPET method, while the method of [9] is standalone.

6 Conclusion

We presented an abstract interpretation technique to automatically build a warp-level CFG for GPU kernels. Our method strictly follows the SIMT semantics as implemented in the Nvidia Pascal GPUs. In particular, it handles the possibility of thread divergence and its impact on the warp-level CFG. Part of our analysis focuses on the representation of

agreement between threads of the same warp on a given value (e.g. a register value) to filter out divergence when we can statically prove that all threads agree on the branch to take. We performed an evaluation on the Rodinia benchmark, and highlighted the importance of the agreement analysis.

In the future we will improve our analysis by supporting the calls to kernels from other kernels, and by adapting classical analyses (e.g. loop bound analysis) to our framework. We will also work on our understanding of the microarchitecture of GPU targets in order to derive precise durations for the blocks of the CFG.

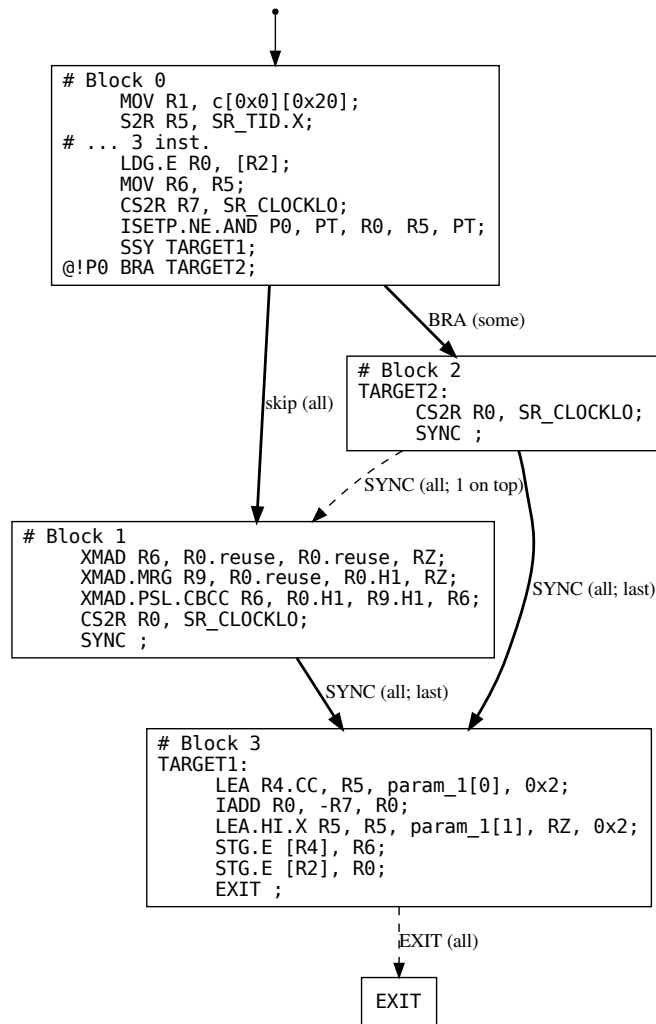
References

- 1 Sebastian Altmeyer and Claire Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *J. Syst. Archit.*, 57(7):707–719, 2011. doi:10.1016/j.sysarc.2010.08.006.
- 2 T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.
- 3 Gogul Balakrishnan and Thomas W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010. doi:10.1145/1749608.1749612.
- 4 Kostiantyn Berezovskyi, Konstantinos Bletsas, and Björn Andersson. Makespan Computation for GPU Threads Running on a Single Streaming Multiprocessor. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 277–286, July 2012. ISSN: 2377-5998. doi:10.1109/ECRTS.2012.16.
- 5 Adam Betts and Alastair Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202, July 2013. ISSN: 2377-5998. doi:10.1109/ECRTS.2013.29.
- 6 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993. doi:10.1007/BFb0039704.
- 7 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009. doi:10.1109/IISWC.2009.5306797.
- 8 Robert I. Davis, Sebastian Altmeyer, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real Time Syst.*, 54(3):607–661, 2018. doi:10.1007/s11241-017-9285-4.
- 9 Vesa Hirvisalo. On Static Timing Analysis of GPU Kernels. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASICS)*, pages 43–52, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 2190-6807. doi:10.4230/OASICS.WCET.2014.43.
- 10 Yijie Huangfu and Wei Zhang. Static WCET Analysis of GPUs with Predictable Warp Scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 101–108, May 2017. ISSN: 2375-5261. doi:10.1109/ISORC.2017.24.
- 11 Rémi Meunier, Thomas Carle, and Thierry Monteil. Correctness and Efficiency Criteria for the Multi-Phase Task Model. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2022.9.

- 12 I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

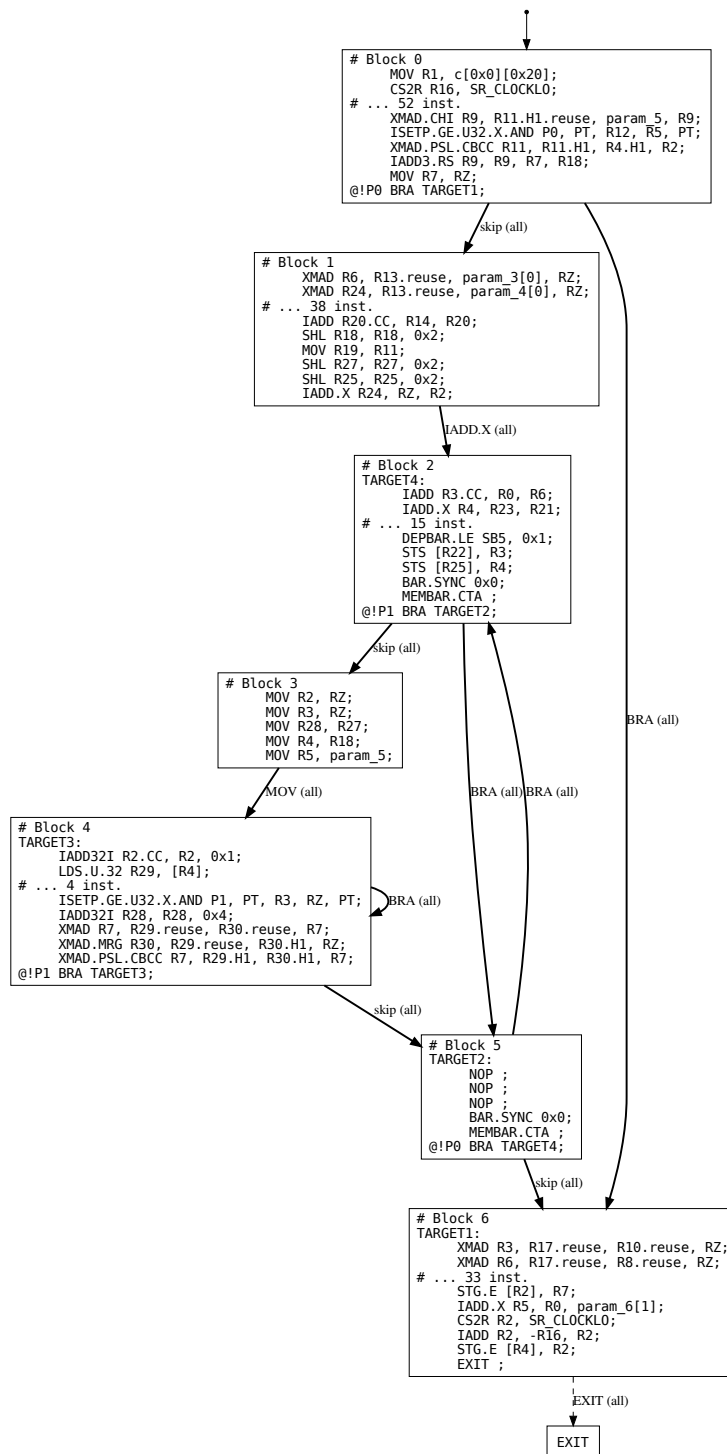
A Appendix

A.1 If-then-else example



■ **Figure 2** Control flow graph of a program containing a non-trivial if-then-else. The dashed control edge is not part of the source control flow.

A.2 Regular loops example



■ **Figure 3** Control flow of a tiled matrix multiplication. Without agreement analysis, the exit instruction in block 6 would have an edge to every potential divergence.

Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators

Alban Gruin  

IRIT – Univ. Toulouse 3 – CNRS, France

Thomas Carle  

IRIT – Univ. Toulouse 3 – CNRS, France

Christine Rochange  

IRIT – Univ. Toulouse 3 – CNRS, France

Pascal Sainrat  

IRIT – Univ. Toulouse 3 – CNRS, France

Abstract

We propose a workflow to help find errors in the processor models that are used to prove their timing predictability. Recently, several papers have modeled processor cores using formal models that represent how instructions progress through the pipeline in each execution cycle. However, such models grow with the complexity of the cores and they are built by hand, using a description of the core, usually the HDL-level code. Such a task is error-prone, and verifying that the model actually captures the core's timing behavior is required, otherwise the proofs become useless. Our workflow simulates the execution of benchmark applications using the HDL specification of a core in order to extract timing information as well as other relevant information (e.g. cache miss events, branch mispredictions). This information is used to replay the execution in a simulator of the core timing model, and to determine whether or not the model accurately represents the execution timing of the instructions. To avoid writing the simulator by hand for each new core, or new variation of a core, we developed a compiler that translates the timing model of a core into a C++ program. We evaluated our approach on the open source MINOTAuR core and we show how it enabled us to detect and correct errors in its model.

2012 ACM Subject Classification Hardware → Simulation and emulation; Hardware → Equivalence checking; Hardware → Safety critical systems

Keywords and phrases Processor model, timing predictability, simulator generation

Digital Object Identifier 10.4230/OASICS.WCET.2023.2

Funding This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the ProTiPP (ANR-22-CE25-0004) project.

1 Introduction

Ensuring the schedulability of real-time systems requires knowing the worst-case execution time (WCET) of each critical task. Deriving such a WCET for tasks running on multicore processors is particularly challenging because tasks running in parallel on separate cores may request accesses to shared hardware components (e.g. shared cache, memory bus or controller) at the same time. This makes the WCET of tasks dependent on their execution context and would require a cycle accurate model of the execution of the whole task set, which is untractable in practice. In order to reduce the complexity of the multi-core WCET analysis, the community has adopted the compositional approach [10], in which the WCET of each task is a composition of its worst-case duration in isolation and of a context-related penalty that is computed as part of an interference analysis.



© Alban Gruin, Thomas Carle, Christine Rochange, and Pascal Sainrat;
licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 2; pp. 2:1–2:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, this approach can be safely applied only if the cores are not prone to so-called timing anomalies [14] i.e. situations in which a local worst case (e.g. a cache miss) does not lead to the global worst case (i.e. the WCET of the analyzed task). Proving the absence of timing anomalies in a core requires a formal timing model that expresses all the details of the inner workings of the core [11, 8, 13]. This model is usually produced manually from the VHDL or SystemVerilog specification of the core, which is particularly complex, time-consuming and error-prone. As such, the formal model is the weak point of the chain, as the applicability of the proofs relies on the fact that the model strictly reflects the behavior of the processor as it is implemented in hardware.

In this paper, we present a methodology to obtain a better level of confidence on the correctness of such formal models, by simulating the timing behavior of the formal model of the processor using instruction traces obtained from the execution of programs on the target processor. We applied our work on the open source MINOTAuR core [8], a processor derived from the RISC-V CVA6 core [16] which was proven to be timing-anomaly free. Our methodology has allowed us to uncover small mistakes in the formal model of MINOTAuR and to fix them. Although this technique does not guarantee correctness, it allows us to gain a higher confidence in the model. To facilitate the implementation of our methodology, we also propose a model description language and an automatic model simulator generator.

The paper is organized as follows. In Section 2, we present the main lines of our model of the MINOTAuR processor. Our validation workflow is introduced in Section 3 and is evaluated on our model in Section 4. In Section 5, we show how model simulators can be automatically generated from the description of a processor model. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Background

In [8], we introduced MINOTAuR, a 6-stage in-order RISC-V core of moderate complexity. MINOTAuR is a timing-anomaly-free version of the CVA6 core [16]. Its monotonicity was proven using a model similar to the one proposed for the SIC processor [11]. This model is expressed in the logic of predicates. It describes the progression of an instruction in the pipeline of the processor, depending on various factors, such as its kind, memory dependencies, or data dependencies. The model is reproduced on Figure 1.

In the first part, it describes the basic structure of the pipeline: name and order of the stages, latency of an instruction cycle in each stage, next stage of an instruction depending on its kind, etc. In each execution cycle c , an instruction i is in a stage $c.stg(i)$, and has a counter $c.cnt(i)$ that indicates the remaining processing time of the instruction in this stage. The $cycle(c)(i)$ function describes what happens to instruction i at the end of cycle c : either the instruction is ready to advance to the next stage and the next stage is ready to process it, in which case the instruction advances, or the instruction remains in its current stage, in which case its processing counter is decreased by one. When the counter reaches 0, the instruction is considered processed in its current stage.

In the second part of the model, the $c.ready(i)$ predicate describes whether an instruction is ready to advance to the next stage in the next cycle: in the general case its processing counter must be equal to 0 and it must be the oldest instruction in the stage. Depending on the stage the instruction currently resides in, additional constraints may apply. For example, in the issue stage, an instruction is not ready if it has a pending data dependency. Potential branch misprediction is accounted for by the $pwrong(i)$ predicate. We refer the interested reader to the original MINOTAuR paper for a comprehensive description of the model.

$$\begin{aligned}
\mathcal{S} &:= \{pre, PC, IF, ID, IS, ALU, MUL_1, MUL_2, DIV, LSU, LU, SU, CSR, CO, ST, post\} \\
pre \sqsubseteq_S PC \sqsubseteq_S IF \sqsubseteq_S ID \sqsubseteq_S IS \sqsubseteq_S \{ALU, MUL_1, LSU, CSR, DIV\} \sqsubseteq_S \{MUL_2, LU, SU\} \sqsubseteq_S CO \sqsubseteq_S ST \sqsubseteq_S post \\
cycle(c)(i) &:= \begin{cases} (c.nstg(i), c.nlat(i)) & : c.ready(i) \wedge c.free(c.nstg(i)) \\ (c.stg(i), c.ncnt(i)) & : otherwise \end{cases} \\
c.nlat(i) &:= \begin{cases} memlat_f(i) & : c.nstg(i) = IF \wedge \neg ichit(i) \\ memlat_d(i) & : (c.nstg(i) = LU \wedge \neg dchit(i)) \\ & \vee c.nstg(i) = ST \\ exlat(i) & : c.nstg(i) = DIV \\ 0 & : otherwise \end{cases} \quad c.ncnt(i) := \begin{cases} c.cnt(i) - 1 & : c.cnt(i) > 0 \\ 0 & : otherwise \end{cases} \\
c.nstg'(i) &:= \begin{cases} PC & : c.stg(i) = pre \\ IF & : c.stg(i) = PC \\ ID & : c.stg(i) = IF \\ IS & : c.stg(i) = ID \\ LSU & : c.stg(i) = IS \wedge opc(i) \in \{load, store, atomic\} \\ LU & : c.stg(i) = LSU \wedge opc(i) = load \\ SU & : c.stg(i) = LSU \wedge opc(i) \in \{store, atomic\} \\ MUL_1 & : c.stg(i) = IS \wedge opc(i) = mul \\ MUL_2 & : c.stg(i) = MUL_1 \\ DIV & : c.stg(i) = IS \wedge opc(i) = div \\ CSR & : c.stg(i) = IS \wedge opc(i) = csr \\ ALU & : c.stg(i) = IS \wedge opc(i) \notin \{load, store, atomic, mul, div, csr\} \\ CO & : c.stg(i) \in \{ALU, MUL_2, DIV, CSR, LU, SU\} \\ ST & : c.stg(i) = CO \wedge opc(i) \in \{store, atomic\} \\ post & : (c.stg(i) = CO \wedge opc(i) \notin \{store, atomic\}) \vee (c.stg(i) = ST) \end{cases} \\
lstg(op) &:= \begin{cases} LU & : op = load \\ ST & : op = store \\ ST & : op = atomic \\ IS & : op = mul \\ DIV & : op = div \\ CO & : op = csr \\ ALU & : op = branch \end{cases} \quad c.nstg(i) := \begin{cases} post & : c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i) \\ c.nstg'(i) & : otherwise \end{cases} \\
c.isnext(s, i) &:= c.stg(i) = s \wedge \forall j < i. c.stg(j) \sqsubseteq_S s \\
c.pending(i, op) &:= \exists j < i. op(j) = op \wedge c(j) \sqsubseteq_P (lstg(op), 0) \\
c.ready(i) &:= (c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i)) \\
&\vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i), i)) \\
&\wedge (c.stg(i) = PC \Rightarrow (ichit(i) \\
&\quad \vee (\neg c.pending(i, branch) \wedge \neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\
&\wedge (c.stg(i) = IS \Rightarrow (opc(i) \notin \{load, store, atomic\} \Rightarrow \neg c.pending(i, csr)) \\
&\quad \wedge (opc(i) \in \{mul, div\} \Rightarrow \neg c.pending(i, div)) \\
&\quad \wedge (\forall j < i. dep(i, j) \Rightarrow c.stg(j) \sqsubseteq_S CO)) \\
&\wedge (c.stg(i) = LSU \Rightarrow (opc(i) \in \{store, atomic\} \wedge \neg c.pending(i, atomic)) \\
&\quad \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\
c.free(s) &:= s \in \{ALU, MUL_1, CSR, MUL_2, CO, post\} \\
&\vee (s \in \{IF, IS, LSU, SU\} \wedge c.slot(s)) \\
&\vee (s \in \{PC, ID, DIV, LU, ST\} \wedge (\neg \exists j. c.stg(j) = s) \vee (\exists j. c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j)))) \\
&\vee (\exists i. c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch)) \\
c.slot(IF) &:= ((\#\{j \mid c.stg(j) = IF\} < fq_size) \vee c.free(ID)) \wedge \forall j. c.stg(j) = IF \Rightarrow c.cnt(j) = 0 \\
c.slot(IS) &:= \#\{j \mid IS \sqsubseteq_S c.stg(j) \sqsubseteq_S CO\} < iq_size \vee (\exists j'. c.isnext(CO, j') \wedge c.ready(j') \wedge (opc(j') \in \{store, atomic\} \Rightarrow c.free(ST))) \\
c.slot(SU) &:= \#\{j \mid opc(j) = store \wedge LSU \sqsubseteq_S c.stg(j) \sqsubseteq_S post\} < sq_size \vee \exists j'. c(j') = (ST, 0) \\
c.slot(LSU) &:= \#\{j \mid c.stg(j) = LSU\} < mq_size \\
&\vee (\exists j'. c.isnext(LSU, j') \wedge ((opc(j') = load \wedge c.free(LU)) \vee (opc(j') \in \{store, atomic\} \wedge c.free(SU))))
\end{aligned}$$

■ **Figure 1** Model of the MINOTAuR core, described in predicate logic, taken from [8].

In the third part, the $c.free(s)$ predicate describes whether a pipeline stage will be free in the next cycle, i.e. whether it can accept a new instruction. As the MINOTAuR core features instructions queues, notably in its issue stage, we also have a feature to count instructions in a stage, and a predicate, $c.slot(s)$, that indicates whether there will be a slot available for an instruction in the next cycle.

Some predicates (e.g. $ichit(i)$, $dep(i, j)$, $prwrong(i)$) and latencies (e.g. $memlat_d(i)$) remain opaque: in order to simplify the model, how their value is obtained is not expressed in the model. For example, $ichit(i)$ is true iff the fetch of instruction i leads to an instruction cache hit. Computing this value for a given instruction sequence would require adding the description of the cache to the model. Instead, the proofs cover both possibilities for the value of $ichit(i)$, and the actual model of the cache is not required.

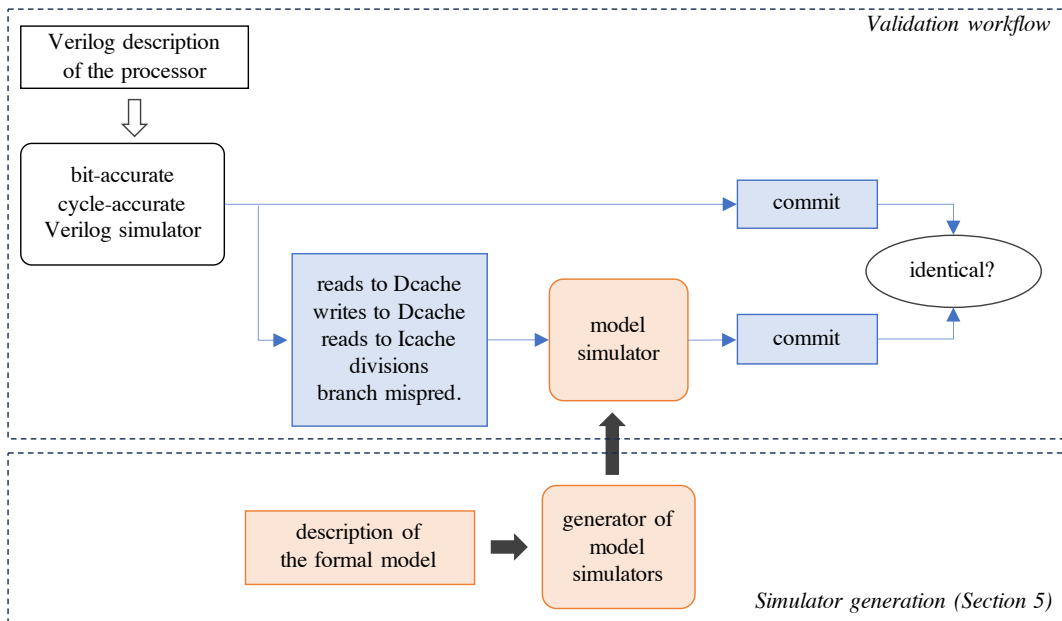
3 Validation workflow

Our validation workflow relies on a simulator of the formal processor model: it is a C++ implementation of the $cycle(c)(i)$ function from this formal model, and thus it only focuses on the timing aspects of the execution.

Figure 2 (upper part) displays the validation workflow. We use a bit-accurate cycle-accurate simulator generated from the Verilog description of the processor and simulate the execution of a benchmark application. For this simulation, we extract the information corresponding to the opaque predicates in our model, allowing us to replay the execution in our simulator. We obtain the following set of traces: reads to the instruction cache, reads to the data cache, writes to the data cache, divisions, and finally, committed instructions. These traces are obtained by adding probes to the SystemVerilog design of the core¹.

The execution is then replayed in the model simulator, that extracts the predicate and latency values from the traces, and generates itself a trace of committed instructions.

Finally, we compare the commit traces from the SystemVerilog simulator and from the model simulator: if they are identical, that means that the model accurately describes the timing behavior of the core for this benchmark. Otherwise, it indicates that there is an error in the model. In that case, the trace can help in narrowing down the search of the error.



■ **Figure 2** Overview of our validation workflow.

Traces formats

The traces are stored in a simple, plain-text format, with one event (i.e. one cache access, or one division, or one branch misprediction) per line.

¹ For cores whose SystemVerilog design is unavailable, we are currently looking at the feasibility of using a hardware probe e.g. Lauterbach debugging probe.

Instruction cache reads trace. An event consists of 5 fields: whether the request is valid or not (i.e. it has not been cancelled by the frontend due to a branch misprediction), start cycle, read address, end cycle, and instruction binary code. The address of the instruction allows its identification all along its progression in the pipeline, and the binary code is necessary to send the instruction to the correct functional unit during the simulation, as well as to track instruction dependencies (predicate $dep(i, j)$). The start and end cycles determine the $ichit(i)$ predicate and the $memlat_f(i)$ latency. Listing 1 shows 3 lines of an instruction cache trace.

Data cache reads trace. An event contains 3 fields: validity of the request, start and end cycles. This is enough to determine the $dchit(i)$ predicate and the $memlat_d(i)$ latency. Other information, such as the address of the access, are not needed for the model simulation.

Data cache writes and divisions traces. They both have 2 fields per line: start and end cycle of the operation. They determine the $dchit(i)$ predicate and $memlat_d(i)$ and $exlat_d(i)$ latencies.

Branch misprediction trace. An event only contains the cycle at which a branch is determined to be mispredicted by the ALU. This is used to set $pwrong(i)$ for all the younger instructions residing in the pipeline at the time the branch reaches the ALU.

Commit trace. In MINOTAuR, a trace of committed instructions is written by default when simulating. It contains a lot of information, such as the commit time and cycle, the instruction address, the opcode and the decoded instruction, or register values. For our purpose, we only need to extract the commit cycle and the address of the instruction. Hence, the commit traces written by our model simulator contain only these two fields.

■ **Listing 1** Extract of the instruction cache reads trace for the CoreMark benchmark. From left to right: validity, start cycle, read address, end cycle, instruction binary code.

```

1 1      282 00000810      286 7b241073
2 1      287 00000814      291 7b351073
3 1      292 00000818      296 00000517

```

4 Evaluation

In this section, we discuss the results obtained by applying our methodology to the processor, i.e., what issues we found in the model, and the limitations of the methodology.

We began by implementing new tracers to the description of the core (in addition to the existing commit tracer) in order to generate all the aforementioned traces. We then ran the CoreMark benchmark and the full TACLe benchmark suite [7] under QuestaSim², a cycle-accurate SystemVerilog simulator, to obtain the traces required by the model simulator.

² <https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/>

4.1 Issues found in MINOTAuR's model and solutions

During our experiments, we found issues in the formal model of MINOTAuR, sometimes related to confusion or misunderstanding of the behavior or structure of the pipeline, and sometimes due to a misinterpretation of the formal logic used in the model. We present 3 of them in the remainder of this section.

Issue 1. One of the most important issues we found using our validation workflow, was related to data dependencies. In function $c.ready(j)$, our model states that an instruction is ready (i.e. can progress) when it has no pending data dependency, or when the instruction it depends on is in or after the commit stage. While this definition is sufficient for read-after-write (RaW) dependencies, it does not cover most write-after-write (WaW) dependencies. Actually, most RaW dependencies can be resolved as soon as the oldest instruction has completed its execution and reached stage CO: the result will be forwarded to the newer instruction. One exception to this principle are CSR³ read instructions: the read is done only when the instruction is committed. In the case of WaW dependencies, the newer instruction has to wait for the oldest instruction to be committed before it can be issued. To solve this, we replaced the $dep()$ predicate with two predicates, one for WaW hazards, and another for RaW hazards (resp. $dep_{WaW}()$ and $dep_{RaW}()$). The dependency check becomes:

$$\begin{aligned} \forall j < i. (dep_{WaW}(i, j) \Rightarrow c.stg(j) \sqsupseteq_S CO) \\ \wedge (dep_{RaW}(i, j) \Rightarrow ((opc(j) = csr \wedge c.stg(j) \sqsupseteq_S CO) \vee (c.stg(j) \sqsupseteq_S CO))) \end{aligned}$$

Issue 2. We also found an inconsistency related to the data cache in the LU stage. The Verilog simulator shows that, whenever a cache miss occurs in the LU, no instruction is processed in the cycle that follows the end of the access. This was not represented in the model, and to account for this, we changed the definition of the $c.free()$ function. The relevant part of the model is the following:

$$\begin{aligned} s \in \{PC, ID, DIV, LU, ST\} \wedge ((\neg \exists j. c.stg(j) = s) \\ \vee (\exists j. c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j)))) \end{aligned}$$

which states that the LU is free if there is no instruction in the stage, or if the instruction in the LU exits the stage in the next cycle. Instead, the stage can be special cased to the following:

$$\begin{aligned} s = LU \wedge ((\neg \exists j. c.stg(j) = LU) \\ \vee (\exists j. c.stg(j) = LU \wedge c.ready(j) \wedge c.free(c.nstg(j)) \wedge dchit(j))) \end{aligned}$$

The relevant change is highlighted. It means that, if there is an instruction in the LU, the stage will be free if the current instruction is a hit, but not if it is a miss.

Issue 3. Another inconsistency concerns the behavior of instructions in the issue stage: the model specifies that all instructions, except loads, stores and atomics, will not be issued in the presence of an uncommitted CSR instruction. In reality, this does not happen (this error was due to misunderstanding when reading the SystemVerilog code), and the part of the expression that is highlighted below can be removed:

$$\begin{aligned} c.stg(i) = IS \Rightarrow (opc(i) \notin \{load, store, atomic\} \Rightarrow \neg c.pending(i, csr)) \\ \wedge (opc(i) \in \{mul, div\} \Rightarrow \neg c.pending(i, div)) \wedge (\forall j < i. dep(i, j)) \end{aligned}$$

³ Control and Status Register

Validation summary. After correcting the model, we were able to run all our benchmarks and found no difference between the commit trace generated by the Verilog simulator and the one obtained by our model simulator.

4.2 Limitations of our methodology

Our validation workflow is based on testing and thus does not provide any guarantee that the chosen benchmarks cover all the possible errors in the model. However, it helped us find and correct a few errors in our model, and thus gain confidence in the revised model.

Additionally, we emphasize that applying our validation workflow with the same benchmarks as those used for performance evaluation purposes guarantees (once all the discrepancies have been fixed) that the performance results have been obtained with a model that reflects the processor's behavior accurately.

5 Automatic generation of timing simulators

One major shortcoming of our method is having to write a simulator that corresponds to the model of the core: this task is error-prone and must be done each time a new core is considered. To ease up the transition from the formal model described in predicate logic, like the one in Figure 1, to an efficient C++ simulator, we designed a domain-specific description language that stays as close as possible to the predicate logic formulas of the model, as well as a compiler written in OCaml. Listing 6 gives the formal definition of our language in EBNF.

It is a functional language similar to both the logic language used in the SIC [11] and MINOTAuR [8] models, and, to some extent, to OCaml. It features some basic data types (integers, booleans, lists, and tuples), with the possibility to define custom enumerations, and optionally, to define an order on the elements of the enumeration. Basic constructs such as simple pattern matching are also available. The types of variables and functions are inferred by the compiler, using a Hindley-Milner type system.

In the remainder of this section, we present key features of our language, using examples taken from the MINOTAuR model on Figure 1, and later explain the compilation process in more details.

5.1 Description of the language

Our language allows the definition of custom enumerations with the `set` or `type` keywords, as well as partial orders on their elements. This can be used to reproduce the definition of the pipeline and instruction kinds in the beginning in our model:

■ **Listing 2** Declaration of the stages and opcodes of the MINOTAuR core.

```

1 set stage = | Pre | PC | IF | ID | IS | ALU | MUL1 | MUL2
2   | DIV | LSU | LU | SU | CSR | CO | ST | Post
3
4 order stage as s = Pre < PC < IF < ID < IS < {ALU, MUL1, LSU, CSR, DIV} <
5   {MUL2, LU, SU} < CO < ST < Post
6
7 type opcode = | Nop | Alu | Mul | Div | Load | Store
8   | Atomic | Branch | Csr | Fence | FenceI | Unknown

```

The `order` keyword marks the beginning of the declaration, and the `as` keyword is used to define a suffix for the `<`, `<=`, `>`, and `>=` operators. Here, to compare two stages, one will have to use the `<s`, `<=s`, `>s`, and `>=s` operators, respectively. Elements between curly braces

are given the same level: $IS <_s ALU$ and $IS <_s LSU$ are true, but $ALU <_s LSU$ and $LSU <_s ALU$ are false, for instance. But, even though ALU and LSU are different, $ALU <=_s LSU$ and $LSU <=_s ALU$ are true.

One can declare variables and functions with the `let` keyword in our language. They can be recursive and co-recursive, as this is required to implement the $c.free(s)$ and $c.slot(s)$ predicates. The user must take care of providing a base case for them. In our case, a recursive call is made to $c.free(c.nstg(j))$, which will eventually be equal to $c.free(post)$. As an example, here is our implementation of the $c.ready(i)$ function:

■ **Listing 3** Implementation of the $c.ready(i)$ function in our language.

```

1 let ready(opc, limit c, i, pwrong) =
2   (stg(c, i) <> Pre /\ !pending(opc, c, i, Branch) /\ pwrong)
3   \/ (cnt(c, i) = 0 /\ isnext(c, stg(c, i), i) /\
4     (stg(c, i) = IS ->
5       (opc[i] in {Mul, Div} -> !pending(opc, c, i, Div))
6       /\ (forall j in c, (j < i -> !dep(opc, c, i, j))))
7     /\ (stg(c, i) = LSU ->
8       (opc[i] in {Store, Atomic} /\ !pending(opc, c, i, Atomic))
9       \/ (opc[i] = Load /\ !pending(opc, c, i, Atomic))))

```

The syntax of the language is very similar to the predicates logic used by the model of MINOTAuR. This example demonstrates multiple features in our language: the multiple comparators and logic connectors, and lists.

Lists are essential in our language to represent traces. Here, `opc` and `c` are lists, used to represent the opcode and the state of an instruction, respectively. To access a specific element, one can use the `[]` operator, as seen on Line 8. It is not advised to use it with a constant or an *ad-hoc* variable, as it may be out-of-bounds. Instead, one can use the `forall`, `exists`, and `#{}` constructs to scan lists. The indices they generate are guaranteed to be valid. Table 1 lists the builtin functions working on lists, as well as their logic equivalent.

Lists exists in two kinds: “normal” lists, and “limited” lists. This distinction exists to allow the code generator to specify bounds for the `forall`, `exists`, and count functions. This allows to load only parts of the traces in memory if the generator elects to do so. The exact bounds are hidden and cannot be manipulated in our language. The only way to obtain a “limited” list is by adding an annotation on a parameter, like on `c` in the example above.

The bounds of a “limited” list cannot be manipulated directly from our language.

It is possible to downcast a “limited” list to a regular list, in which case the bounds will be dropped. It is also possible to upcast a regular list to a “limited” list. In this case, the bounds will cover the whole list.

■ **Table 1** Builtin functions for lists.

Construct	Logic equivalent
<code>c[i], index(c, i)</code>	$c[i]$
<code>forall i in c, (...)</code>	$\forall i \dots$
<code>exists i in c, (...)</code>	$\exists i \dots$
<code>#{j in c ...}</code>	$\#\{j \dots\}$

5.2 Compilation of the model

We developed a compiler for our language, which takes care of type checking and code generation in C++. This compiler only generates code for the portions of the model that vary with each processor (the `ready`, `free`, `nstg`, `lstg` and `slot` functions). The high-level functions (`cycle`, `nlat`) are hard-coded in a template file that is used for all processors, as well as the I/O functions that read the traces, decode the instructions (find their kind, latencies and dependencies) and set the opaque predicate values.

In our language, all constructs are expressions, which is not the case of conditional blocks and for loops in C++. Thus, boolean expressions, like in Listing 3, may be translated as pure expressions or as a sequence of conditional blocks when loop constructs are required. Loop constructs are generated when using the `forall`, `exists`, or `#{}` list builtins, as shown on Listing 4.

■ **Listing 4** Example of translation of the `forall`, `exists`, and `#{}` constructs.

```

1 // forall j in c, stg(c, j) = s -> j < i
2 bool tmp0 {true};
3 for (unsigned int j {0}; j < c.size(); ++j)
4     tmp0 = tmp0 && (j < i || !(stg(c, j) == s));
5
6 // exists j in c, stg(c, j) = s -> j < i
7 bool tmp1 {false};
8 for (unsigned int j {0}; j < c.size(); ++j)
9     tmp1 = tmp1 || (j < i || !(stg(c, j) == s));
10
11 // #{j in c / stg(c, j) = s -> j < i}
12 unsigned int tmp2 {0};
13 for (unsigned int j {0}; j < c.size(); ++j) {
14     if (j < i || !(stg(c, j) == s))
15         ++tmp2;
16 }

```

Notice that the `j` index has a value ranging from 0 to the size of the list minus 1. This is the behavior for regular lists. For limited lists, the index would have a value within the bounds. When a function takes a limited list as a parameter, the bounds are added automatically by the compiler in the prototype and at each call site. The prototype of the `ready` function is generated as shown on Listing 5. The compiler does not modify the bounds, thus it is up to the interface between the generated code and the rest of the simulator to handle them.

■ **Listing 5** C++ prototype of the `ready` function.

```

1 bool ready(opcode opc, std::vector<stage, unsigned int> c, unsigned int
    ↪ c_start, unsigned int c_end, unsigned int i, bool pwrong);

```

As seen on Listing 5, lists are compiled to standard C++ vectors, which requires to load the entire trace in memory. This works for the majority of programs of the TACLe benchmark suite, but some, such as `ammunition`, were so big that their traces did not fit in the memory of our test machine. To alleviate this problem, our simulator loads chunks of the trace files as they are needed through a special vector type. To benefit from this mechanism, the generated code has to be modified manually for now.

Sets (resp. orders), as shown in Listing 2, are compiled to enumerations (resp. operator overload) in C++. When an order is defined for a set, a function converting each element to an integer is generated. The lowest element (e.g. `Pre`) is assigned the value 0, with this value growing for each level. The overloaded operators then convert each element to an integer, and compare the results.

6 Related Work

Various kinds of formal models of the timing behavior of processors have been proposed in the literature. These models are designed to support the estimation of worst-case execution times [12, 6, 2, 9] or the proof of properties, such as the absence of timing anomalies [11, 4, 1]. They describe how an instruction or a code sequence flows through the pipeline using so-called execution graphs [12, 2], timed automata [6, 9], a transition system [4] or a set of logic predicates [11]. These models are usually designed by hand, either from the processor user manual or from its HDL description.

A few papers consider automatically deriving the processor’s model from HDL code [15, 3] to alleviate the risk of errors in the model but they only provide preliminary solutions.

In [5], the authors model the functional and timing behavior of simple processor using the L3 domain specific language, translate it in a HOL4 version and confront it to the execution on the real processor of short code snippets. The processor we consider in this paper implements complex mechanisms (e.g. speculative execution) that are not addressed in their paper.

7 Conclusion and Future Work

We introduced a workflow to validate that a timing model of a processor corresponds to the actual execution timing on the real processor. This workflow is based on a simulator of the model that replays traces obtained by executing (or simulating) the execution of a benchmark on the actual processor. Since the model focuses only on the timing aspects of the core, so does our simulator. This allows the simulator to be very simple, compared to a functional simulator. By comparing the original execution trace to the one obtained with the model simulator, we can detect errors in the model and correct them. We applied this methodology to the model of the open-source MINOTAuR core and were able to find and correct several issues in the model, using the CoreMark and TACLe benches. In order to facilitate the use of this workflow, we also presented a compiler that automatically generates the model simulator from a language that is very close to the predicate language in which the cores are described.

Being based on testing, our workflow does not provide a guarantee that all mistakes have been corrected in a model, but it still allows to increase the confidence one can have in a given model. As part of future work, we envision to extend our workflow to more complex cores featuring out-of-order execution, and to use our description language to automatically generate proofs of the absence of timing anomalies in Coq. We could thus generate proofs and simulators from the exact same model.

References

- 1 Mihail Asavoae, Belgacem Ben Hedia, and Mathieu Jan. Formal executable models for automatic detection of timing anomalies. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis, WCET*, 2018.
- 2 Zhenyu Bai, Hugues Cassé, Thomas Carle, and Christine Rochange. Computing execution times with execution decision diagrams in the presence of out-of-order resources. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.
- 3 Samira Ait Bensaïd, Mihail Asavoae, Farhat Thabet, and Mathieu Jan. Work in progress: Automatic construction of pipeline datapaths from high-level HDL code. In *28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

- 4 Benjamin Binder, Mihail Asavoae, Belgacem Ben Hedia, Florian Brandner, and Mathieu Jan. Is this still normal? putting definitions of timing anomalies to the test. In *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021.
- 5 Brian Campbell and Ian Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In *Formal Methods for Industrial Critical Systems*, 2014.
- 6 Franck Cassez, Pablo Gonzalez de Aledo, and Peter Gjøøl Jensen. Wuppaal: Computation of worst-case execution-time for binary programs with uppaal. In *Models, Algorithms, Logics and Tools*, pages 560–577. Springer, 2017.
- 7 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 2:1–2:10, 2016.
- 8 Alban Gruin, Thomas Carle, Christine Rochange, Hugues Cassé, and Pascal Sainrat. MINOTAuR: A timing predictable RISC-V core featuring speculative execution. *IEEE Transactions on Computers*, 72(1):183–195, 2022.
- 9 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.
- 10 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*, pages 299–308, 2016.
- 11 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: a provably timing-predictable pipelined processor core. *Real-Time Systems*, 56(2):207–245, 2020.
- 12 Xianfeng Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *25th IEEE International Real-Time Systems Symposium*, 2004.
- 13 Michael Platzer and Peter Puschner. Vicuna: a timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- 14 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.
- 15 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. *SIGPLAN Not.*, 45(4):67–76, April 2010.
- 16 Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

A Appendix

A.1 Formal definition of our language

■ **Listing 6** Formal definition of our language in EBNF.

```

1 digit = "0" | ... | "9";
2 letter = "a" | ... | "z";
3 id = letter, {"a" | ... | "z" | "0" | ... | "9" | "_" | "-"};
4 number = ["-" | "+"], {digit};
5 subset = "{", {id, "|"}, id, "}";
6
7 description = {type_declaration | order_declaration |
  ↪ function_declaration};

```

2:12 Automatic Processor Simulator Generation

```
8 type_declaration = ("type" | "set"), id, "=", ["|"], {id, "|"}, id;
9 order_declaration = "order", id, "as", id, "=", {(id, subset), "<"}, (id,
  ↪ subset);
10
11 param = id | ("limit" id);
12 param_list = {param, ",", param};
13 function_contents = id, ["(", param_list, ")"], "=", expression;
14 function_declaration = "let", ["rec"], function_contents, {"and",
  ↪ function_contents};
15
16 comparison = expression, ("/\" | "\/" | "->" | "=" | "<>" | (("<" | ">"),
  ↪ ["="], [id])), expression;
17 inside = expression, ["not"], "in", subset;
18 negation = ("!" | "not"), expression;
19 forall = ("forall" | "exists"), id, "in", expression, ",", "(" ,
  ↪ expression, ")";
20 count = "#{", id, "in", expression, "|", expression, "}";
21 list_access = expression, "[", expression, "]";
22 tuple = "(", {expression, ","}, expression, ")";
23 priority = "(", expression, ")";
24 funcall = expression, "(", [{"limited"}, expression, ","}, [{"limited"},
  ↪ expression], ")";
25 immediate = id | "true" | "false" | number | tuple;
26 match = "match", expression, "with", {"|"}, immediate, "->", expression,
  ↪ ["|", "-", "->"], expression, "end";
27 if = "if", expression, "then", expression, "else", expression;
28 expression = function_declaration | comparison | inside | negation |
  ↪ forall | count | list_access | priority | funcall | immediate |
  ↪ match | if;
```

Exploring iGPU Memory Interference Response to L2 Cache Locking

Alfonso Mascareñas González ✉ 
ISAE-SUPAERO, Université de Toulouse, France

Jean-Baptiste Chaudron ✉ 
ISAE-SUPAERO, Université de Toulouse, France

Régine Leconte ✉
ISAE-SUPAERO, Université de Toulouse, France

Youcef Bouchebaba ✉
ONERA, Université de Toulouse, France

David Doose ✉
ONERA, Université de Toulouse, France

Abstract

The demand of parallel execution in real-time embedded applications has motivated the integration of GPUs as processing accelerators on SoCs (System-on-Chip) embedded architectures, often leading to CPU-iGPU architectures. In the safety-critical domain, it is paramount to ensure that the execution deadlines of critical tasks are not exceeded. To ease the analysis of this kind of tasks, we can make their worst-case execution time more predictable. One way to achieve this is by mitigating or controlling the memory interference generated by the concurrent execution of tasks through the application of a series of techniques (e.g., cache partitioning, bank partitioning, cache locking, bandwidth regulation). Originally, these were applied to CPUs, and more recently, to GPUs as well. In this work, we focus on the hardware-based L2 cache locking on iGPUs as memory interference mitigation mechanism. We are interested in evaluating its capacity for reducing the worst-case and the average-case execution time in different scenarios. Our measurement-based analysis has been carried out on the NVIDIA's Jetson AGX Orin 64 GB MPSoC, making use of four representative benchmarks (data resetting, 2D convolution, 3D convolution and matrix upsampling).

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

Keywords and phrases iGPU, cache locking, real-time, memory interference

Digital Object Identifier 10.4230/OASICS.WCET.2023.3

Supplementary Material *Software (Source Code)*: <https://github.com/ISAE-PRISE/gcalasy>
archived at `swh:1:dir:2a28897e208903f22dfa9e8fc6ba0eba569ff2ce`

Funding This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE N° 2019 65 0090004707501).

1 Introduction

Since the last decade, we have been observing an increase in popularity of MPSoCs where CPUs work together with integrated Graphic Processing Units (iGPUs). This is the solution that manufacturers have devised to address the need for higher execution parallelism in real-time embedded systems for autonomous machines. These MPSoCs might be used for safety-critical real-time applications, meaning that the execution deadline of the tasks must be satisfied to avoid fatal outcomes. Therefore, it is of extreme importance to perform Worst-Case Execution Time (WCET) analysis of all tasks making up the overall application. The memory interference among the tasks concurrently executing is one of the main reasons



© Alfonso Mascareñas González, Jean-Baptiste Chaudron, Régine Leconte, Youcef Bouchebaba, and David Doose;
licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 3; pp. 3:1–3:11



OpenAccess Series in Informatics

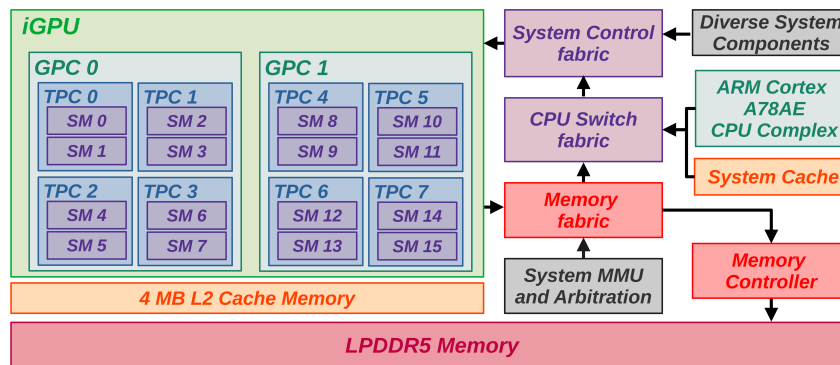
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for missing deadlines. Traditionally, this type of interference is avoided or mitigated through spatial-temporal isolation (e.g., bandwidth regulation) and memory partitioning techniques (e.g., cache partitioning, cache locking, bank partitioning) [5].

In this work, we focus on the iGPU, more specifically, on its on-chip shared L2 cache memory. This cache is shared across all the multiple Streaming Multiprocessors (SMs) composing the GPU, and hence, being susceptible to data evictions. To avoid undesired data removal, specially of the data we consider critical or persistent, we apply the cache locking technique. Our objective is to assess the capacity of cache locking for: (1) mitigating the inter-SM interference within one GPU application, (2) mitigating the inter-SM interference produced by a variable number of non-critical GPU applications on one critical GPU application and (3) mitigating the Low-Power Double Data Rate 5 (LPDDR5) memory interference on one GPU application. To do so, we make use of realistic benchmarks (data resetting, 2D matrix convolution, 3D matrix convolution and matrix upsampling), from which we study their WCET and Average Case Execution Time (ACET). The measurement-based analysis of the L2 cache locking technique is carried out on the Jetson AGX Orin 64 GB [11], a heterogeneous MPSoC by NVIDIA which allows performing cache locking using hardware-specific features.

2 Related Work

Memory interference has been a constant problem for the real-time community since the introduction of the first multicore platforms. Since then, researchers have proposed a series of techniques to deal with them [5]. Initially, these techniques were aimed to CPUs but adapted variations have been also implemented on GPUs. To begin with, let us consider the bandwidth regulators, which control the amount of requests that other cores can issue. Work [1] presents a software mechanism that implements bandwidth regulators and GPU protection for CPU-GPU architectures (see [16] for CPU approach). We can continue with the cache and bank partitioning techniques which consist in dividing these two shared resources so that each division is private to a processing core. The former technique offers protection against forced cache evictions by concurrent tasks while the latter partially or totally removes the costly intra-bank interference on the main memory. These two techniques are applied on GPUs in work [4] (see [15] for CPU), which makes use of reverse-engineering and page coloring for performing these partitions. Furthermore, work [4] makes use of SM partitioning, i.e., compute kernels belonging to the same application execute on dedicated SMs. Task and memory mapping can also be used for reducing the memory interference. For instance, on a CPU and DSP context, work [7] performs task-core and core-bank mapping for minimizing interference while considering other optimization objectives. Finally, we have the cache locking technique which consists in keeping loaded data in the cache, preventing it from being evicted. Caches can be fully locked (i.e., the whole cache is used for locked data) or partially locked (i.e., a part of the cache is for locked data and another for cacheable data). The locking can be made statically (i.e., locking is done at boot time and remains unchanged) or dynamically (i.e., locking changes during run-time as function of the needs). Generally, static locking offers more predictability while dynamic locking, which can also be predictable, is more performing [6]. The cache locking technique has been commonly applied for increasing the predictability, performance and energy efficiency of CPUs (see [8]). On GPUs, work [13] uses a simulator to evaluate the impact of cache locking. A comparison between cache partitioning and cache locking for CPU-GPU architectures is made by work [14], also via simulations. In contrast to these previous works, ours aims to study this technique on a real MPSoC incorporating an iGPU with L2 cache locking capabilities.



■ **Figure 1** Ampere architecture GPU in Jetson AGX Orin 64 GB.

3 Target Platform

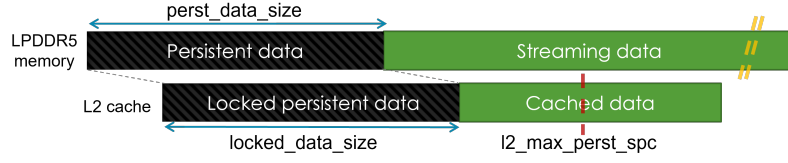
3.1 Introduction

The heterogeneous multicore platform used in this work is the Jetson AGX Orin 64 GB by NVIDIA [11]. It is made up of 12 Cortex-A78AE grouped in clusters of 4 and an Ampere architecture iGPU. The iGPU is composed of 2048 cores distributed among 16 SMs. Each SM has a dedicated L1 cache memory of 192 KB and, all of them, share an L2 cache memory of 4 MB. Internally, the SMs are subdivided in four processing blocks, each of them with their own registers (64 KB in total), an instruction cache and 32 cores (32 threads/clock). As seen in Figure 1, a pair of SMs makes up a Texture Processing Cluster (TPC), and 4 TPCs compose a Graphic Processing Cluster (GPC). There are 2 GPCs in the Orin MPSoC. Computing kernels are executed as grids of blocks of threads, each block being composed of threads. On Jetson AGX Orin, a thread block supports up to 1024 threads. As long as there are enough resources in a SM, multiple blocks of threads can be run by the same SM. We point out two important constraints, the number of threads and blocks supported by a SM. On this platform, the former is limited to 1536 and the latter to 16 [12].

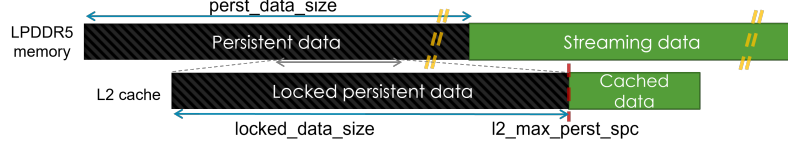
3.2 L2 cache locking hardware mechanism

In NVIDIA GPUs with a compute capability of 8.0 or above, L2 cache locking can be done through hardware [9, 10]. This is achieved by: (1) tagging a contiguous region of global memory as persistent (NVIDIA term for lockable) and (2) setting the amount of space from the L2 cache capacity to use for this persistent memory. If the amount of data tagged as persistent is lower than the reserved persistent data space in the L2 cache (i.e., $perst_data_size < l2_perst_spc$), then normal data (termed “streaming data” by NVIDIA) can occupy this space. We must avoid having more persistent data than space reserved for it in the L2 cache (i.e., $perst_data_size > l2_perst_spc$), as L2 cache lines thrashing will take place [9], and therefore, no effective cache locking. To avoid this issue, there are two possibilities. The first one consists in tuning the L2 cache *hitRatio* parameter, which is used to determine the percentage of persistent data to be locked in the cache. The second option is to limit the amount of persistent data to the L2 cache reserved space (i.e., $perst_data_size \leq l2_perst_spc$) even if this means that some persistent data is left unlocked. We make use of the latter option as the first one implies a non-deterministic factor (i.e., the portion of memory to be locked in is unknown), and therefore, is not suitable for predictability analysis. Another constraint to respect has to do with the maximum persistent

3:4 Exploring iGPU Memory Interference Response to L2 Cache Locking



(a) $perst_data_size \leq l2_perst_spc$.



(b) $perst_data_size > l2_perst_spc$.

■ **Figure 2** Mapping persistent data to L2 cache.

L2 cache data space that we are allowed to set (i.e., $l2_perst_spc \leq l2_max_perst_spc$). $l2_max_perst_spc$ is 3 MB for the Orin MPSoC, meaning that there is at least a minimum of 1 MB of cacheable space to be used for any kind of data. Our L2 cache locking space is computed according to Equation 1, where $perst_data_size$ is the size of persistent data to lock and $l2_perst_spc$ the amount of L2 cache space reserved for locking.

$$locked_data_size = \min(perst_data_size, \min(l2_perst_spc, l2_max_perst_spc)) \quad (1)$$

Figure 2 shows two examples which depict how persistent and non-persistent (streaming) data is placed on the L2 cache. In both cases (Figures 2a and 2b), we assume that the reserved L2 cache space for locking ($l2_perst_spc$) is set to the maximum space allowed ($l2_perst_spc = l2_max_perst_spc = 3\text{ MB}$). In Figure 2a, the streaming data in the LPDDR is bigger than the one shown in the L2 cache. In this example, not all the reserved lockable space is used as the persistent data ($perst_data_size$) fits inside the cache with less space ($perst_data_size < l2_perst_spc$). Therefore, the locked data size is $perst_data_size$. In contrast, Figure 2b supposes the case where the persistent data exceeds the reserved lockable space ($perst_data_size > l2_perst_spc$). Note that the persistent and streaming data in the LPDDR is bigger than the one shown in the L2 cache. In this case, the locked data size is $l2_perst_spc$.

4 Evaluation

The L2 cache locking is evaluated in three scenarios described in Sections 4.1, 4.2 and 4.3. The customized benchmarks used for the evaluation are based on the data resetting (Equation 2), 2D convolution (Equation 3), 3D convolution (Equation 4) and upsampling operations (Equation 5):

$$B[j] = A[j \% m] \quad (2)$$

where $A = \forall(a_i) \in \mathbb{R}^m$ and $B = \forall(b_j) \in \mathbb{R}^p$. Vector A is the data to protect with the cache locking mechanism.

$$C[i, j]^{(k)} = \sum_{u=0}^p \sum_{v=0}^q A \left[i + u - \lfloor \frac{p}{2} \rfloor, j + v - \lfloor \frac{p}{2} \rfloor \right] \cdot B[u, v]^{(k)} \quad (3)$$

with $A = \forall(a_{ij}) \in \mathbb{R}^{m \times n}$, $B^{(k)} = \forall(b_{ij}) \in \mathbb{R}^{p \times q}$ and $C^{(k)} = \forall(c_{ij}) \in \mathbb{R}^{m \times n}$. The superindex k is the matrices identification for each convolution compute kernel launch ($k \in \{0, 1, \dots, K-1\}$). We consider matrix A as persistent data.

$$C[i, j, k]^{(k)} = \sum_{u=0}^p \sum_{v=0}^q \sum_{w=0}^l A \left[i + u - \lfloor \frac{p}{2} \rfloor, j + v - \lfloor \frac{p}{2} \rfloor, w \right] \cdot B[u, v, w]^{(k)} \quad (4)$$

with $A = \forall(a_{ijk}) \in \mathbb{R}^{m \times n \times l}$, $B = \forall(b_{ijk}) \in \mathbb{R}^{p \times q \times l}$ and $C = \forall(c_{ijk}) \in \mathbb{R}^{m \times n \times l}$. The superindex k is the matrices identification for each convolution compute kernel launch ($k \in \{0, 1, \dots, K-1\}$). We consider tensor A as persistent data.

$$B = (A)_{\uparrow L} \quad (5)$$

where $A = \forall(a_{ij}) \in \mathbb{R}^{m \times n}$ and $B = \forall(b_{ij}) \in \mathbb{R}^{p \times q}$, L is the upsampling value and A is the data to lock in the L2 cache.

The execution time of these benchmarks is computed through CUDA events recording, process that consists is retrieving a timestamp of the start and stop events before and after the execution of the benchmark respectively. For each observed WCET and ACET result provided in Sections 4.1, 4.2 and 4.3, 10^5 measurements were considered. Initial measurements affected by the load of the CUDA driver are considered statistical outliers, and hence, discarded.

4.1 Scenario 1: SM interference of a single compute kernel

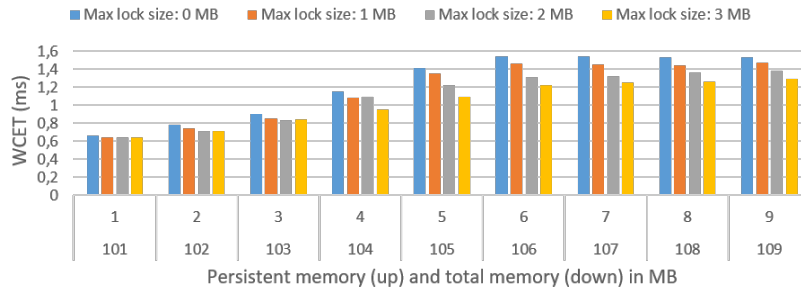
The objective of this scenario is to test the capabilities of the L2 cache locking for mitigating the inter-SM interference when executing a single application, i.e., how the application interferes with itself by making use of all the available SMs. In this scenario, the graphs with the results show the response of a benchmark using the measured WCET and ACET metrics when fixing the lockable space (*l2_perst_spc*) to 0 MB (no locking), 1 MB, 2 MB and 3 MB (*l2_max_pesrt_spc*) represented using blue, orange, grey and amber colors respectively. The X axis indicates the amount of persistent and total data (persistent plus streaming) resulting from varying the size of the data set. The Y axis represents the metric value under analysis in milliseconds. The benchmarks used for testing are the ones described in Equations 2, 3, and 5. The results are illustrated in Figures 3, 4 and 5 respectively.

The data resetting benchmark yields great results. On average, the WCET is reduced by 14.64% with a maximum reduction of 23% (Figure 3a) and the ACET is reduced by 15.45% with a maximum reduction of 24.54% (Figure 3b) for a 3 MB cache lock. For a single application inter-SM interference, the previous gains can be considered high. This is due to the high reuse of specific data (persistent data). By locking this data, the number of loads to the LPDDR are reduced, mainly performing stores.

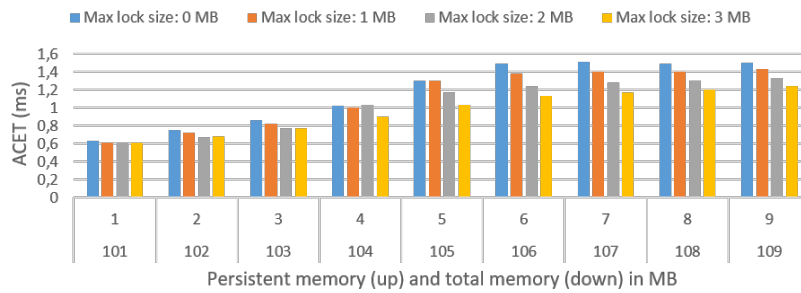
More typical operations are offered by the 2D convolution and upsampling benchmarks. On average, the 2D convolution has its WCET reduced a 5.55% with a maximum reduction of 10.24% (Figure 4a) and the ACET is reduced a 6.7% with a maximum reduction of 24.54% (Figure 4b) for a 3 MB cache lock. The upsampling sees an average WCET reduction of 15.67% with a maximum reduction of 29.15% (Figure 5a) and an average ACET reduction of 13.66% with a maximum reduction of 28.35% (Figure 5b) for a 3 MB cache lock.

Overall, we can see that the L2 cache locking manages to reduce the WCET and ACET. In this sense, the 2D convolution is the one benefiting the least and the data reset and upsampling the most. For the three benchmarks, note that significant execution time drops are achieved even when the persistent data of the benchmark exceeds the L2 lockable space.

3:6 Exploring iGPU Memory Interference Response to L2 Cache Locking

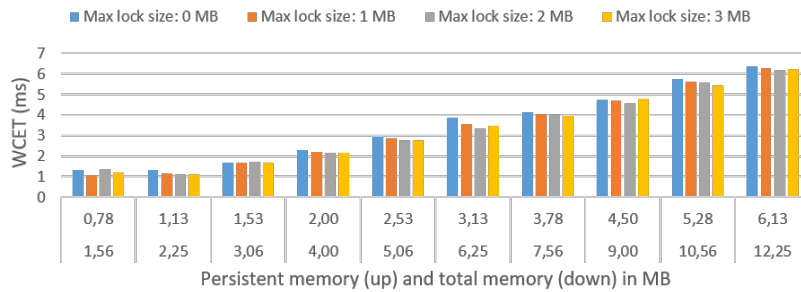


(a) Measured worst-case execution time.

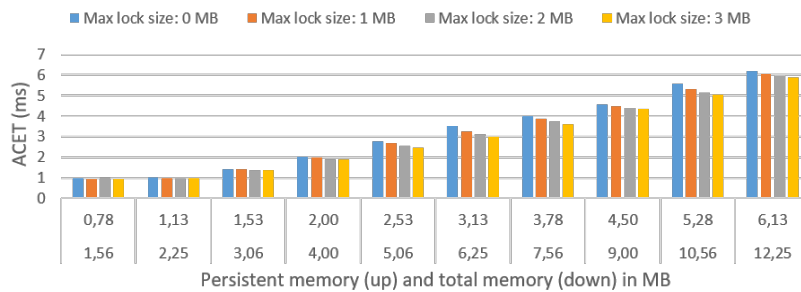


(b) Average-case execution time.

■ Figure 3 Data reset benchmark behavior.

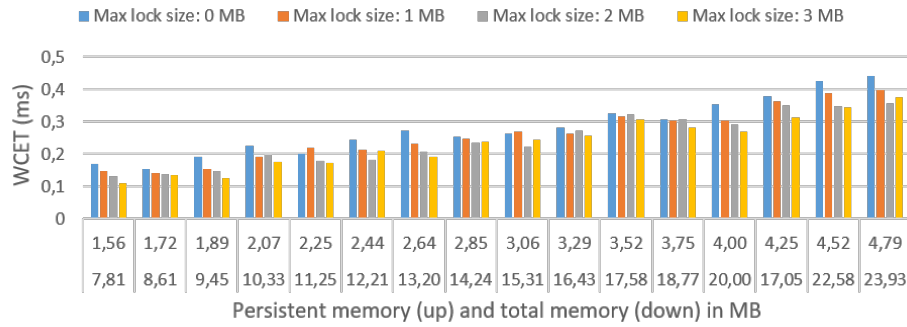


(a) Measured worst-case execution time.

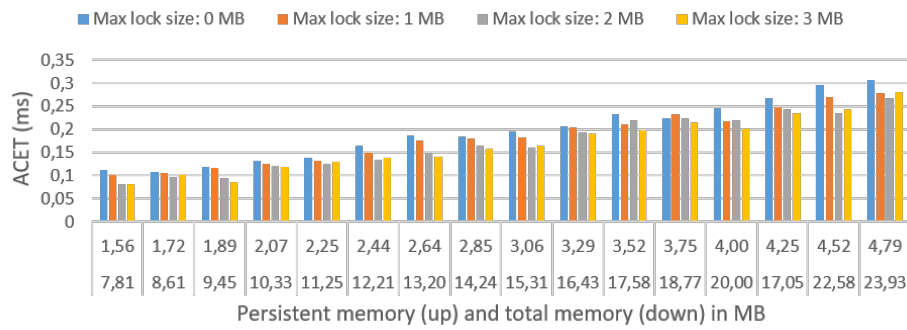


(b) Average-case execution time.

■ Figure 4 2D convolution benchmark behavior.



(a) Measured worst-case execution time.



(b) Average-case execution time.

■ **Figure 5** Matrix upsampling benchmark behavior.

4.2 Scenario 2: SM interference from non-critical compute kernels

In this scenario, we seek to analyze the capacity of the L2 cache locking for isolating the SMs used by a critical application from SM interference produced by non-critical applications. To perform this study, we make use of SM partitioning in order to place together on a SM the thread blocks belonging to the same application. The partitioning is done by exploiting the threads per SM constraint (alternatively we can use the blocks per SM constraint). The interfering non-critical applications are executed as persistent kernels, i.e., the compute kernel re-execute the non-critical applications without launching from host again. The data resetting benchmark (Equation 2) plays the role of the interfering applications.

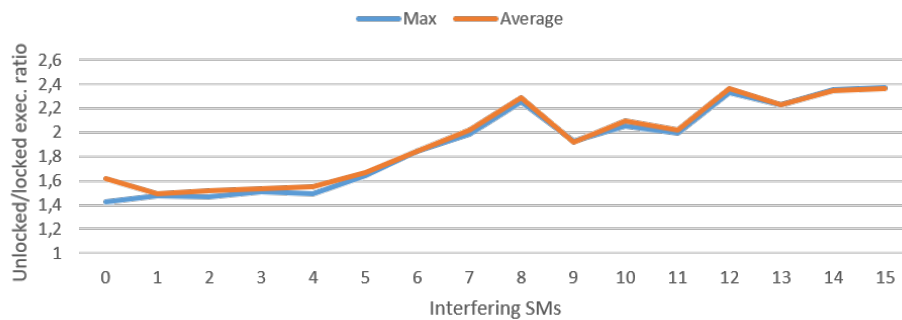
Figures 6a, 6b,6c show the WCET and ACET gain due to the L2 cache locking (Y axis) as function of the interfering number of SMs (X axis). The benchmarks used for testing are those described in Equations 2, 3, and 4.

The measures taken from the data resetting benchmark (Figure 6a) indicate a clear benefit for the WCET and ACET metrics. The performance gain is more notable as the number of interfering SMs increases. For example, when half of the SMs are used by the critical application, the WCET and ACET are 2.26 and 2.29 times less respectively with respect to the L2 cache non-locking approach. As commented in Section 4.1, these gains are not the most typical ones as this benchmark specially makes use of persistent data whose protection from being evicted yields remarkable results. In this sense, the 2D and 3D convolution benchmarks (Figures 6b,6c) offer more common responses. The ACET of the former benchmark benefits more than its WCET, having time reductions of 13.24% and

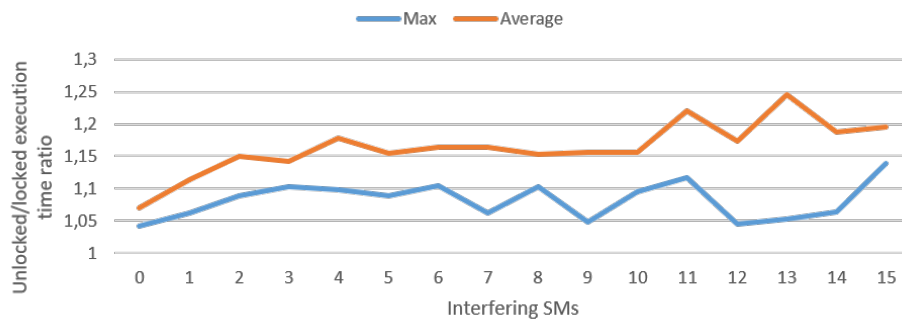
3:8 Exploring iGPU Memory Interference Response to L2 Cache Locking

9.38% respectively when using 8 critical SMs. The WCET and ACET of latter benchmark are similar, progressively benefiting of the L2 cache lock as the number of interfering SMs increases (no significant gains with low number of non-critical SMs). With 8 critical SMs, the WCET and ACET have a 10.14% and 6.8% boost respectively.

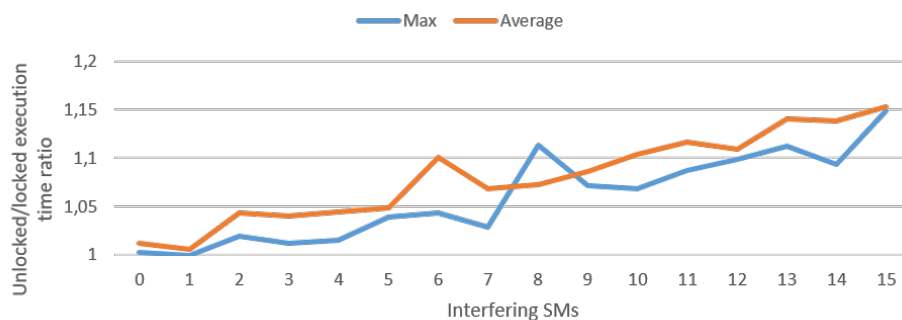
In general, the L2 cache locking serves well for protecting the locked persistent data from other concurrently running applications, leading to performance enhancements of the former. Its effectiveness mainly depends on the design of the applications (critical and non-critical) and the resources used for them (e.g., number of SMs, blocks, threads) but also on the L1 and L2 cache capacity of the iGPU.



(a) Data resetting. Configuration: 3 MB of persistent data and 103 MB of total data.



(b) 2D convolution. Configuration: 2 MB of persistent data and 4 MB of total data.



(c) 3D convolution. Configuration: 4 MB of persistent data and 8.04 MB of total data.

■ **Figure 6** Performance gain under non-critical SM interference led by 3 MB L2 cache locking.

4.3 Scenario 3: LPDDR5 interference from ARM cores

The purpose of the third scenario is to analyze the capacity of the L2 cache locking for mitigating the overheads produced by the contention of the LPDDR memory when stressed by 11 ARM cores. These execute a benchmark based on stores and loads from different array strides, resulting in high level of DDR memory interference. By locking the persistent data, the number of accesses to the LPDDR memory from the iGPU would be reduced, and hence, suffer from less interference. We make use of two locking sizes (none or 3 MB) and different benchmark memory configurations. The benchmarks used are the data resetting (Equation 2), the 2D convolution (Equation 3) and the upsampling (Equation 5). The WCET and ACET response to the L2 cache locking of the two first benchmarks can be seen in Figures 7a and 7b and Figures 8a and 8b respectively.

The data resetting benchmark results show a WCET and ACET improvement when applying the L2 cache lock. Nevertheless, by comparing these results with the homologous of Scenario 1 (Figure 3), we can deduce that the improvement comes from the inter-SM interference reduction rather than from avoiding the LPDDR5 interference as it would be expected. The LPDDR5 memory interference represents, on average, 25.07% and 20.21% of the WCET and ACET respectively. In contrast to the previous benchmark, the 2D convolution benefits from the mitigation of the LPDDR5 interference when applying the L2 cache locking. On average, 60.49% and 36.36% of the WCET and ACET reduction comes from this interference mitigation. On average, the LPDDR5 memory interference is responsible of 26.33% and 18.98% of the WCET and ACET respectively. In the same line, we have observed that for the upsampling benchmark, 65% and 63.16% of the WCET and ACET reduction are due to LPDDR memory interference mitigation.

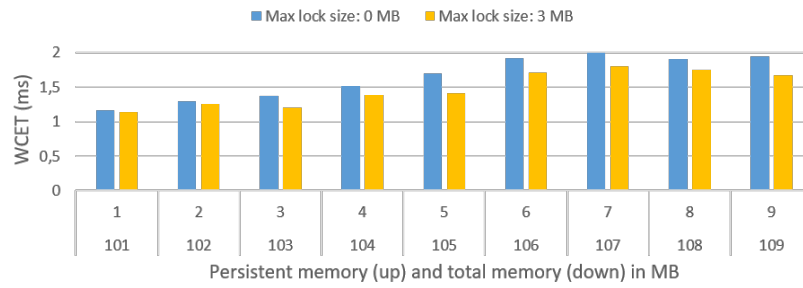
All in all, we have seen that the L2 cache locking is able to reduce the main memory interference. However, as the data resetting benchmark has shown us, this is not always the case, reducing instead the inter-SM interference as in Scenario 1 (Section 4.1).

5 Conclusions and Future work

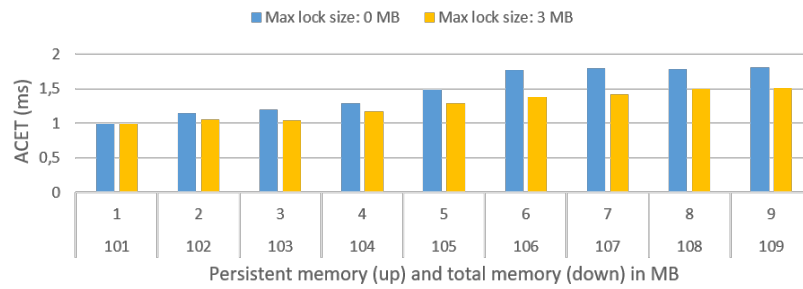
The effectiveness of the L2 cache locking depends on the application to execute, the GPU resources used of it, the amount of persistent and total data used by the application and the GPU L1 and L2 cache capacity. For the used tests and scenarios, we have observed a significant WCET and ACET reduction with many of the configurations we have used. The maximum WCET mitigation of inter-SM interference with one application (Section 4.1) lies between 10.24% and 28.35% depending on the test. For the inter-SM interference with 8 non-critical applications running concurrently (Section 4.2), we observe a maximum WCET reduction that ranges from 9.38% to 55.69%. In the case of the LPDDR interference scenario (Section 4.3), we have have seen maximum performance amelioration between 16.6% and 19.17%.

The benchmarks used in this work are made of a single compute kernel. Therefore, as future work, we would like to extend this study by testing the L2 cache locking for applications composed of chains of compute kernels like in neural network applications (e.g., Resnet [3], Segnet [2]). Besides, there is an aspect that should be studied regarding the L2 cache mechanism. According to NVIDIA, locked data on the L2 cache is automatically unlocked if not used [10]. The conditions under which this reset occurs should be analyzed.

3:10 Exploring iGPU Memory Interference Response to L2 Cache Locking

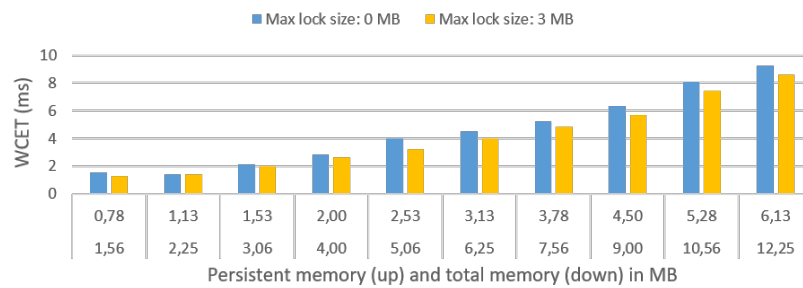


(a) Measured worst-case execution time.

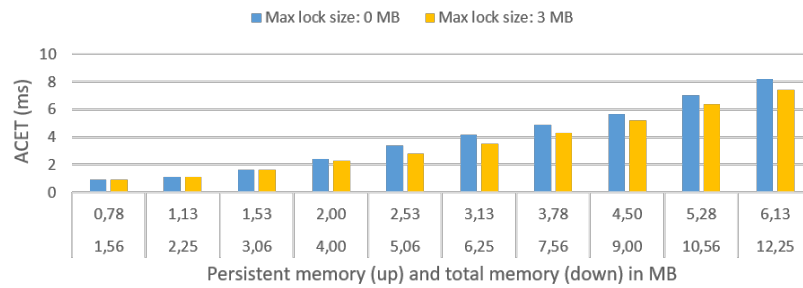


(b) Average-case execution time.

■ **Figure 7** Data resetting behavior under LPDDR5 interference.



(a) Measured worst-case execution time.



(b) Average-case execution time.

■ **Figure 8** 2D convolution behavior under LPDDR5 interference.

References

- 1 Waqar Ali and Heechul Yun. Work-in-progress: Protecting real-time gpu applications on integrated cpu-gpu soc platforms. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 141–144, 2017. doi:10.1109/RTAS.2017.26.
- 2 Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017. doi:10.1109/TPAMI.2016.2644615.
- 3 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.
- 4 Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019. doi:10.1109/RTAS.2019.00011.
- 5 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022. doi:10.1109/ACCESS.2022.3151891.
- 6 Antonio Martí-Campoy, Angel Perles, Francisco Rodríguez-Ballester, and J. Busquets-Mataix. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology*, volume 2, pages 1283–1286 vol.2, June 2003. doi:10.1109/CCECE.2003.1226134.
- 7 Alfonso Mascareñas González, Jean-Baptiste Chaudron, Frédéric Boniol, Youcef Bouchebaba, and Jean-Loup Bussenot. Task and memory mapping optimization for sdram interference minimization on heterogeneous mpsoCs. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE Press, 2022. doi:10.1109/ETFA52439.2022.9921677.
- 8 Sparsh Mittal. A survey of techniques for cache locking. *ACM Transactions on Design Automation of Electronic Systems*, 21(3), May 2016. doi:10.1145/2858792.
- 9 NVIDIA. *CUDA C++ Best Practices Guide*, May 2022.
- 10 NVIDIA. *CUDA C++ Programming Guide*, December 2022.
- 11 NVIDIA. *NVIDIA Orin Series System-on-Chip - TECHNICAL REFERENCE MANUAL*, March 2022.
- 12 NVIDIA. *Ampere Tuning Guide*, February 2023.
- 13 John Picchi and Wei Zhang. Impact of l2 cache locking on gpu performance. In *SoutheastCon 2015*, pages 1–4, 2015. doi:10.1109/SECON.2015.7133036.
- 14 Xin Wang and Wei Zhang. Cache locking vs. partitioning for real-time computing on integrated cpu-gpu processors. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2016. doi:10.1109/PCCC.2016.7820644.
- 15 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014. doi:10.1109/RTAS.2014.6925999.
- 16 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.

Clustering Solutions of Multiobjective Function Inlining Problem

Kateryna Muts  

Hamburg University of Technology, Germany

Heiko Falk  

Hamburg University of Technology, Germany

Abstract

Hard real time-systems are often small devices operating on batteries that must react within a given deadline, so they must satisfy their timing, code size, and energy consumption requirements. Since these three objectives contradict each other, compilers for real-time systems go towards multiobjective optimizations which result in sets of trade-off solutions. A system designer can use the solution sets to choose the most suitable system configuration. Evolutionary algorithms can find trade-off solutions but the solution set might be large which complicates the task of the system designer. We propose to divide the solution set into clusters, so the system designer chooses the most suitable cluster and examines a smaller subset in detail. In contrast to other clustering techniques, our method guarantees that the sizes of all clusters are less than a predefined limit. Our method clusters a set by using any existing clustering method, divides clusters with sizes exceeding the predefined size into smaller clusters, and reduces the number of clusters by merging small clusters. The method guarantees that the final clusters satisfy the size constraint. We demonstrate our approach by considering a well-known compiler-based optimization called function inlining. It substitutes function calls by the function bodies which decreases the execution time and energy consumption of a program but increases its code size.

2012 ACM Subject Classification Information systems → Clustering; Software and its engineering → Compilers; Computer systems organization → Real-time systems; Mathematics of computing → Evolutionary algorithms

Keywords and phrases Clustering, multiobjective optimization, compiler, hard real-time system

Digital Object Identifier 10.4230/OASICS.WCET.2023.4

1 Introduction

Hard real-time systems are computing systems that must react before a given deadline to avoid catastrophic consequences. The Worst-Case Execution Time (WCET) of a program is its worst possible execution time independent of input data. By minimizing WCETs, we can guarantee that hard real-time systems satisfy their timing constraints. Since many embedded systems have small memories and operate on batteries, code size and energy consumption should also be minimized.

Modern compilers offer optimizations [20] that automatically improve code quality by decreasing code size, execution time, or energy consumption. But these three objectives contradict each other, i.e. when a compiler decreases one of them, it usually increases the others. An optimization problem with conflicting objectives is called multiobjective.

To choose the most desirable trade-off between the objectives, the preferences of a system designer must be incorporated into the solution process. If the designer knows the preferences before the solution process, three main approaches exist [5]: (1) all but one of the objectives are placed into constraints; (2) all objectives are combined into a single objective; (3) a decision maker conducts the solution process in direction of the desired solution.



© Kateryna Muts and Heiko Falk;

licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 4; pp. 4:1–4:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

If the designer wants to know all possible trade-offs before choosing the best one, a Pareto set is generated by, e.g. using an evolutionary algorithm [6]. A Pareto set consists of trade-off solutions for which improvement in one objective worsens at least one of the other objectives.

“The magical number seven, plus or minus two” effect [19] says that humans can handle only a limited amount of information simultaneously, so the system designer cannot examine a Pareto set that consists of dozens of solutions. To improve the decision-making process, two main approaches exist:

1. select a small number of solutions that represent the entire Pareto set [25, 14, 11];
2. divide the Pareto set into clusters of small sizes [3].

In this paper, we focus on the second approach: we cluster a solution set of a multiobjective compiler-based optimization into subsets not larger than a given size.

Many existing clustering methods generate a specified number of clusters ignoring the size of each cluster. Due to “the magical number seven, plus or minus two” effect, we want to guarantee that the sizes of clusters are less than a predefined size to simplify the task of the system designer. We propose the following procedure:

1. we cluster a solution set by using an existing clustering method,
2. if the size of a cluster exceeds the predefined size, we divide the cluster into smaller clusters such that the sizes of the new clusters satisfy the size constraint,
3. we reduce the number of clusters by merging small clusters such that new clusters satisfy the size constraint.

We demonstrate the applicability of the proposed clustering method on a well-known compiler-based optimization called function inlining. This optimization substitutes a function call by the body of the callee. It potentially decreases WCET and energy consumption but increases code size, so no single optimal solution exists that minimizes the objectives simultaneously and the problem becomes multiobjective [21].

We organized the paper as follows: Section 2 presents related work, Section 3 describes the concepts of multiobjective optimization problems and introduces the proposed clustering method, Section 4 evaluates the clustering method by applying it to compiler-based optimization called function inlining, and Section 5 gives a conclusion.

2 Related Work

Compiler-based optimizations rarely employ multiobjective methods. Lokuciejewski et al. [16] considered bi-objective problems with WCET, average-case performance, and code size as objectives when searching for optimal compiler optimization sequences. Jadhav and Falk [12] presented a bi-objective static SPM allocation with WCET and energy consumption as objectives. Muts [21] studied a multiobjective function inlining with three objectives: WCET, energy consumption, and code size. Section 4 describes the multiobjective function inlining problem, since we use it to demonstrate the applicability of the clustering method proposed in this paper.

The Pareto front is a set in the objective space that represents the objectives of a Pareto set. To the best of our knowledge, clustering has been never integrated into compiler-based optimizations but it has been used to reduce Pareto fronts of other multiobjective optimizations.

Mattson, Mullur, and Messac [18] proposed a method to reduce a Pareto front such that the reduced set represents its trade-off properties. A designer controls the set size and the degree of practically insignificant trade-offs, which defines how far two solutions should be from each other to keep both of them in the reduced set. The authors used the approach to reduce Pareto fronts of bi- and tri-objective mathematical problems and a physical truss design problem.

Catania et al. [4] aimed to reduce a Pareto front that represents the performance, power, and area of an Application Specific Instruction-set Processor (ASIP) when configuring it, e.g. by setting the size of a cache. The authors used fuzzy *c*-means to partition the Pareto front and kept one solution from each cluster in the reduced set. They used Xie-Beni index [29] to identify the number of clusters.

Ishibuchi, Pang, and Shang [11] used an expected loss function to select a representative subset of the Pareto front. The expected loss function measures the loss when one solution is chosen instead of another, so the final subset minimizes the expected loss. The authors compared the approach to hypervolume-based subset selection methods [14] by considering mathematical test problems.

Kong et al. [13] proposed a clustering-based decision-making method for the multiobjective reservoir operation problem. The authors clustered a Pareto set and its Pareto front in the decision and objective spaces, respectively, to identify solutions in high-density areas of the decision and objective spaces. They selected a compromise solution by using both clustering results.

Li, Wu, and Yang [15] used TOPSIS [30] – the technique for order performance by similarity to an ideal solution – to select solutions from a Pareto front when solving a multiobjective conceptual design problem with product assembly, manufacturing, and cost as objectives. For the TOPSIS method, a designer provides a decision matrix that contains scores of the objectives for each solution of the Pareto front. TOPSIS ranks and selects solutions based on the distances to the positive- and negative-ideal solutions. The authors demonstrated the approach by considering the conceptual design of a centrifugal compressor.

Smedberg and Bandaru [26] developed an interactive decision support system that allows decision makers to visualize Pareto fronts and to study their impact in the decision space. To visualize two-dimensional projections of solutions from high-dimensional objective spaces, the authors implemented radial coordinate visualization, *t*-distributed stochastic neighbour embedding, uniform manifold approximation and projection, scatter plots, and parallel coordinate plots. To select solutions from a Pareto front, the authors implemented reference point-, lasso- and slider-based selections. To extract knowledge about the decision vectors of the selected solutions, the authors implemented two data mining techniques: Flexible Pattern Mining [2] and Simulation-Based Innovization [9] which generate decision rules. To visualize the extracted knowledge, the authors implemented a graph-based technique, where nodes represent decision rules and edges connect the rules such that the combined rule meets the significance thresholds set by a user. They used the system to study benchmark optimization problems with up to 10 objectives and real-world problems with up to six objectives.

The approaches described above extract a subset of a Pareto front that represents the entire Pareto front, and a system designer chooses the final solution from the extracted set. Such approaches might discard solutions that suit most of the system designer's requirements. Our method divides a Pareto front into clusters without discarding any solutions, and the system designer selects first a cluster and then the most suitable solution from the cluster.

Bejarano, Espitia, and Montenegro [3] studied *k*-means and *c*-means fuzzy algorithms in terms of clustering Pareto fronts obtained by solving eight artificial multiobjective problems. The authors clustered the Pareto fronts into 2–6 clusters. Both clustering algorithms produced similar clusters for continuous Pareto fronts but complementary clusters for discontinuous fronts. *K*-Means was slower than fuzzy *c*-means on the considered Pareto fronts. In contrast to our approach, this method ignores the sizes of clusters.

3 Clustering Pareto Front

Many real-world optimization problems deal with conflicting objectives, i.e. when we improve one objective, we degrade the others. Optimization problems with conflicting objectives are called multiobjective. A general *multiobjective minimization problem without constraints* is formulated as follows:

$$\text{minimize } \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})), \quad (1)$$

where $m \in \mathbb{N}$ and $f_i : X \rightarrow \mathbb{R}$ with $X \subset \mathbb{R}^d$ and $i = 1, 2, \dots, m$.

For a minimization problem with m objectives, a decision vector \mathbf{x}_1 *dominates* another vector \mathbf{x}_2 or in symbols $\mathbf{x}_1 \prec \mathbf{x}_2$, if $f_i(\mathbf{x}_1) \leq f_i(\mathbf{x}_2)$ for all $i \in \{1, 2, \dots, m\}$ and there exists $j \in \{1, 2, \dots, m\}$ such that $f_j(\mathbf{x}_1) < f_j(\mathbf{x}_2)$. A solution $\mathbf{x} \in X$ is called *Pareto optimal*, if it is not dominated by any other solution. Any multiobjective optimization problem results in a solution set P called *Pareto set* which consists of trade-off solutions: $P = \{\mathbf{x} \in X : \mathbf{x} \text{ is Pareto optimal}\} \subset \mathbb{R}^d$. The *Pareto front* F represents the subset of the objective space corresponding to the Pareto set: $F = \{\mathbf{f}(\mathbf{x}) : \mathbf{x} \in P\} \subset \mathbb{R}^m$.

Two main approaches exist to solve multiobjective problems: (1) iterative scalarization methods iteratively change their parameter values to generate a Pareto set; (2) evolutionary algorithms evolve a set of solutions - called population - over several iterations - called generations - by using bio-inspired genetic operators. Evolutionary algorithms are commonly used in practice [7], since scalarization methods produce one solution per iteration, whereas evolutionary algorithms generate a set of solutions in each iteration.

After solving a multiobjective optimization problem, a system designer selects a desirable solution from the resulting Pareto front. Since Pareto fronts are usually large and humans can handle only a limited amount of information, clustering techniques simplify the designer's task by dividing the Pareto front into clusters.

Well-known clustering methods, e.g. k-means, generate a specified number of clusters but ignore cluster sizes. We aim to cluster a Pareto front such that the size of each cluster does not exceed a given maximum size.

■ Algorithm 1 Clustering.

Require: Set $S \subset \mathbb{R}^m$, clustering algorithm *Cluster*, maximum cluster size τ , maximum distance $dist_{max}$ between two clusters to be merged

Ensure: Set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset S$, $\bigcup S_i = S$, and $S_i \cap S_j = \emptyset$ for $i \neq j$

1: $n \leftarrow \lceil \frac{|S|}{\tau} \rceil$ ▷ Number of clusters

2: $\mathcal{S} \leftarrow \text{Cluster}(S, n)$

3: $\mathcal{S} \leftarrow \text{RefineClusters}(\mathcal{S}, \text{Cluster}, \tau)$ ▷ Algorithm 2

4: $\mathcal{S} \leftarrow \text{MergeClusters}(\mathcal{S}, \tau, dist_{max})$ ▷ Algorithm 3

Algorithm 1 presents the proposed clustering procedure. The algorithm takes a set S (Pareto front) to be clustered, a clustering algorithm *Cluster*, e.g. k-means, a desired maximum cluster size τ , and a maximum possible distance between two clusters $dist_{max}$ for merging the clusters. The parameter $dist_{max}$ guarantees that two clusters are merged only if they are close enough to each other. The algorithm returns a set of clusters \mathcal{S} . Since many clustering algorithms require a number of clusters as an input, at Line 1, the algorithm computes the number of clusters n based on the size of the set S denoted by $|S|$ and the maximum cluster size τ . The clustering algorithm *Cluster* clusters the set into n clusters. Since the clustering algorithm might generate clusters larger than the maximum size τ ,

at Line 3, the algorithm refines the clusters such that the size of each cluster is not larger than τ . The refinement might generate small clusters with sizes much smaller than τ , so at Line 4, the algorithm merges the small clusters if it is possible.

■ **Algorithm 2** Refine clusters.

Require: Set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset \mathbb{R}^m$, clustering algorithm $Cluster$, maximum cluster size τ

Ensure: Set of refined clusters $\mathcal{R} = \{R_i\}$ with $R_i \subset \mathbb{R}^m, |R_i| \leq \tau$, and $\bigcup R_i = \bigcup S_i$

```

1:  $C_{largest} \leftarrow LargestCluster(\mathcal{S})$ 
2:  $\mathcal{R} \leftarrow \mathcal{S}$ 
3: while  $|C_{largest}| > \tau$  do
4:    $\mathcal{C}_{refined} \leftarrow Cluster(C_{largest}, \lceil \frac{|C_{largest}|}{\tau} \rceil)$ 
5:    $\mathcal{R} \leftarrow \mathcal{R} \setminus \{C_{largest}\} \cup \mathcal{C}_{refined}$  ▷ Update  $\mathcal{R}$ 
6:    $C_{largest} \leftarrow LargestCluster(\mathcal{R})$ 
7: end while

```

Algorithm 2 presents a procedure to refine clusters with sizes larger than τ . It takes a set of clusters \mathcal{S} , a clustering algorithm $Cluster$, and a maximum cluster size τ . The algorithm returns a set of refined clusters \mathcal{R} with sizes of all clusters less than or equal to τ .

At Lines 1 and 2, we denote by $C_{largest}$ the largest cluster in \mathcal{S} and assign the resulting set \mathcal{R} to the original set \mathcal{S} . While the size of the largest cluster $|C_{largest}|$ is larger than the desired size τ , we cluster it in smaller clusters at Line 4. We update the current set \mathcal{R} with the new clusters at Line 5 and get the largest cluster $C_{largest}$ of the updated set \mathcal{R} at Line 6. We repeat the refinement until the sizes of all clusters are less than or equal to τ .

► **Lemma 1.** *If the input set \mathcal{S} and its clusters $S_i \in \mathcal{S}$ are finite, and $0 < \tau < \infty$, Algorithm 2 terminates and returns clusters of size less than or equal to τ .*

Proof. If $|S_i| \leq \tau$ for all $S_i \in \mathcal{S}$, the algorithm terminates and returns the original clusters S_i .

To prove that the algorithm terminates if there exist clusters with sizes greater than τ , we prove that the *while* loop at Lines 3–7 terminates. We denote by \mathcal{W} a set of all clusters of size greater than τ in \mathcal{S} :

$$\mathcal{W} = \{W \in \mathcal{S} : |W| > \tau\} . \quad (2)$$

Since the input set \mathcal{S} is finite, the set \mathcal{W} is finite.

At the first iteration of the *while* loop, $C_{largest} \in \mathcal{W}$ is split into $n = \lceil \frac{|C_{largest}|}{\tau} \rceil < \infty$ clusters at Line 4. We prove by contradiction that the size of at least one of the newly created clusters is less than or equal to τ : if $|C_k^{new}| > \tau$ for all newly created clusters C_k^{new} with $k = 1, 2, \dots, n$, then

$$|C_{largest}| = \sum_{k=1}^n |C_k^{new}| > \sum_{k=1}^n \tau = n \cdot \tau = \left\lceil \frac{|C_{largest}|}{\tau} \right\rceil \cdot \tau \geq |C_{largest}|. \quad (3)$$

The newly created clusters with sizes less than or equal to τ are removed from \mathcal{W} : $\mathcal{W} = \mathcal{W} \setminus \{C_k^{new} : |C_k^{new}| \leq \tau\}$. At the next iteration, the largest cluster from the updated set \mathcal{W} is split into smaller clusters, and clusters with sizes less than or equal to τ are removed from \mathcal{W} . Since the set \mathcal{W} is finite and monotonically decreases in cardinality, we continue until the set \mathcal{W} is empty. It proves that the algorithm terminates.

The set \mathcal{R} is updated at each iteration and

$$\mathcal{R} = \{W \in \mathcal{S} : |W| \leq \tau\} \cup \{W \in \mathcal{S} : |W| > \tau\} = \{W \in \mathcal{S} : |W| \leq \tau\} \cup \mathcal{W} \quad (4)$$

Since the algorithm terminates when $\mathcal{W} = \emptyset$, the final set \mathcal{R} contains clusters of size less than or equal to τ . ◀

■ **Algorithm 3** Merge clusters.

Require: Maximum cluster size τ , set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset \mathbb{R}^m$ and $|S_i| < \tau$, maximum distance $dist_{max}$ between two clusters to be merged

Ensure: Set of clusters $\mathcal{R} = \{R_i\}$ with $R_i \subset \mathbb{R}^m$, $|R_i| \leq \tau$, and $\bigcup R_i = \bigcup S_i$

```

1:  $\mathcal{R} \leftarrow \mathcal{S}$ 
2: for  $C \in \mathcal{R}$  do
3:    $C_{closest} \leftarrow ClosestCluster(C, \mathcal{R})$  ▷  $C_{closest} \neq C$ .
4:   if  $dist(C, C_{closest}) < dist_{max}$  AND  $|C| + |C_{closest}| < \tau$  then
5:      $C_{merged} \leftarrow Merge(C, C_{closest})$ 
6:      $\mathcal{R} \leftarrow \mathcal{R} \setminus \{C, C_{closest}\} \cup \{C_{merged}\}$  ▷ Update  $\mathcal{R}$ 
7:   go to 2
8: end if
9: end for

```

Algorithm 3 presents a procedure to merge small clusters after refinement. It takes a maximum cluster size τ to preserve the desired sizes of clusters, a set of clusters \mathcal{S} , and a maximum distance $dist_{max}$ between two clusters to be merged. The algorithm returns a set of clusters \mathcal{R} with a reduced number of clusters. At Line 1, the algorithm assigns the output set \mathcal{R} to the original set \mathcal{S} . For each cluster, it gets the cluster $C_{closest} \neq C$ closest to the current cluster C . At Line 4, we denote by $dist$ the distance between two sets $P, Q \subset \mathbb{R}^m$ defined as the Euclidean distance between the centroids \mathbf{g}_P and \mathbf{g}_Q of the sets P and Q , respectively. If the distance between the clusters is smaller than $dist_{max}$ and the sum of the cluster sizes is less than the maximum cluster size τ , we merge the clusters and update the current set \mathcal{R} . When the set \mathcal{R} is updated, the *for* loop iterates over the updated set \mathcal{R} .

► **Lemma 2.** *If $0 < \tau < \infty$, the input set \mathcal{S} is finite, and $|S_i| < \tau$ for all $S_i \in \mathcal{S}$, Algorithm 3 terminates and returns clusters of size less than or equal to τ .*

Proof. To prove that the algorithm terminates, we prove that the *for* loop terminates. The algorithm starts with \mathcal{S} assigned to \mathcal{R} . Since \mathcal{S} is finite, \mathcal{R} is finite and $|R| < \tau$ for all $R \in \mathcal{R}$.

Case 1. If for all $C \in \mathcal{R}$, the closest cluster $C_{closest}$ does not satisfy the conditions at Line 4, the loop terminates after $|\mathcal{R}| < \infty$ iterations.

Case 2. If there exists $C \in \mathcal{R}$ such that its closest cluster $C_{closest}$ satisfies the conditions at Line 4, the clusters C and $C_{closest}$ are merged, the size of the merged cluster is less than τ , and the set \mathcal{R} is updated by substituting the clusters C and $C_{closest}$ by one merged cluster. We denote by \mathcal{R}_1 the updated set \mathcal{R} . The new set \mathcal{R}_1 satisfies the following conditions:

- $|\mathcal{R}_1| = |\mathcal{R}| - 1 < \infty$,
- $|R| < \tau$ for all $R \in \mathcal{R}_1$.

The set \mathcal{R}_1 satisfies either *Case 1* or *Case 2* with $\mathcal{R} = \mathcal{R}_1$. *Case 1* terminates the algorithm and *Case 2* generates a new set of clusters which we denote by \mathcal{R}_2 . By repeating the procedure, we generate a sequence of finite sets $\{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_s\}$. The sets monotonically decrease in cardinality and $|R| < \tau$ for all $R \in \mathcal{R}_i, i = 1, 2, \dots, s$, which proves the lemma. ◀

We considered one of the most widely used clustering algorithm k-means [17] as a clustering method in Algorithms 1 and 2. We also compared the results of k-means with the results of two other clustering methods: spectral clustering [8, 22] and agglomerative clustering with the complete linkage criterion [27].

4 Results

We demonstrate the applicability of the proposed clustering method on a well-known compiler-based optimization called *function inlining* [20].

Performing function inlining, a compiler replaces a function call with the body of the function: it stores the inputs of the function call to local variables, removes the function call, inserts the function body into the code, and removes the return instruction.

Function inlining can decrease WCET and energy consumption since it

- removes call and return instructions which smooths pipeline behaviour;
- reduces parameter handling;
- enables more possibilities for subsequent optimizations, e.g. redundant path elimination or constant propagation which tightens WCET and energy consumption estimations.

Function inlining combined with other optimizations, e.g. redundant path elimination, might decrease code size but it often increases code size due to duplicated function bodies.

The multiobjective function inlining problem with WCET, energy consumption, and code size as objectives is formulated as follows [21]:

- *decision space*: $X = [0, 1]^d$, where the dimension d is equal to the total number of function calls in a program. A decision vector $\mathbf{x} = (x_1, x_2, \dots, x_d)$ is defined as follows:

$$x_i = \begin{cases} 1, & \text{the function is inlined at the function call } i, \\ 0, & \text{otherwise .} \end{cases} \quad (5)$$

- *objective function*: $\mathbf{f} = (\text{WCET}, \text{Energy Consumption}, \text{Code Size})$
- *optimization problem*: $\min_{\mathbf{x} \in X} \mathbf{f}(\mathbf{x})$.

After solving the multiobjective function inlining problem by the evolutionary algorithm MBPOA [28] and getting a Pareto front, we cluster the solutions as described in the previous section.

We use the WCC compiler framework [10] for the ARM Cortex-M0 microcontroller. We computed WCET and energy consumption by AbsInt's aiT and EnergyAnalyser 20.10i [1] and code size by WCC. We ran all evaluations on a computer with Dual CPU Intel Core i7-5600U, RAM 15 GB, 2 CPU cores, and 2.60 GHz CPU frequency. We implemented Algorithms 1, 2, and 3 in Python 3.10.

Function inlining shows the most significant improvement of a final executable in combination with other optimizations, e.g. constant propagation and dead code elimination, so we perform all evaluations with the compiler optimization level `O2`. The Cortex-M0 microcontroller lacks a hardware floating-point unit, so we use the WCC software math library to tackle this issue. We considered functions of the floating-point library as candidates for inlining. We used benchmarks from EEMBC benchmark suite [24] where MBPOA resulted in more than 10 solutions because otherwise, the clustering problem is trivial. We assumed that a benchmark fits into the Flash memory of the architecture.

When clustering a Pareto front found by MBPOA, we consider only meaningful solutions, i.e. solutions with decreasing WCET or energy consumption compared to the original program. Table 1 lists the total number of solutions and the number of meaningful solutions. For all benchmarks, except `bitmnp01`, the applied constraint insignificantly reduced the Pareto fronts.

■ **Table 1** The number of solutions when solving the multiobjective function inlining problem by evolutionary algorithm MBPOA.

	Benchmark												
Number of solutions	a2time01	aifirf01	basefp01	bitmnp01	cacheb01	canrdr01	des	iirfft01	pntrch01	puwmod01	rspeed01	tblock01	ttsprk01
Total	29	38	13	41	21	42	15	26	43	91	17	13	96
Meaningful solutions	29	37	13	11	20	41	15	26	43	91	16	13	96

We used Algorithm 1 to cluster the Pareto fronts. We utilized k-means, spectral clustering, and agglomerative clustering as input clustering methods in the algorithm. We used the implementation of the clustering methods provided by the tool `scikit-learn` [23]. We preserved the `scikit`'s default values provided for the methods. We set maximum cluster size $\tau = 7$ in Algorithm 1 due to “the magical number seven, plus or minus two” effect [19], and maximum distance $dist_{max}$ was computed as described in the following remark:

► **Remark 3.** We did not pass maximum distance $dist_{max}$ as input to Algorithms 1 and 3 but computed it in Algorithm 3 as follows:

$$dist_{max} = \frac{d_{max}}{n - 1} , \quad (6)$$

where n is the number of clusters in the input set \mathcal{S} and d_{max} is the maximum distance between two points from the union of sets $S_i \in \mathcal{S}$:

$$d_{max} = \max_{p, q \in \cup S_i} \|p - q\| . \quad (7)$$

Figure 1 shows cluster sizes as box plots after each stage of Algorithm 1 shown in x-axes: original clustering, refinement of large clusters, and merger of small clusters. The figure shows the results when using agglomerative clustering, k-means, and spectral clustering in Algorithm 1. The numbers on the medians show the total number of clusters. E.g. for the benchmark `canrdr01`, the original agglomerative clustering resulted in six clusters with the largest cluster size 16; after refinement, the algorithm produced 9 clusters with the maximum cluster size 7; and after merging, the algorithm returned 8 clusters. As expected, refinement reduces the cluster sizes to the desired value but increases the number of clusters, whereas merging small clusters may reduce the number of clusters.

If we choose the best clustering method to be used in Algorithm 1 based on the number of final clusters: fewer clusters are better, k-means outperforms the two other clustering methods. For all benchmarks, Algorithm 1 with k-means resulted in the same or smaller number of clusters than when it was combined with the two other clustering methods.

Table 1 shows that six benchmarks `bitmnp01`, `des`, `rspeed`, `basefp01`, `cacheb01`, and `tblock` have a few meaningful solutions compared to the remaining benchmarks. For these benchmarks, Figure 1 shows that

- the refinement and merging stages of Algorithm 1 preserved the original clusters (`bitmnp01`, `des`, `rspeed`) or
- refinement was required to generate clusters of the desired sizes (`basefp01`, `cacheb01`, and `tblock`), whereas merging was useless.

For the remaining seven benchmarks with more solutions to be clustered, the refinement stage was necessary to generate clusters of the desired sizes. For all these benchmarks, except `aifirf01`, the merging stage reduced the number of final clusters for at least one of the

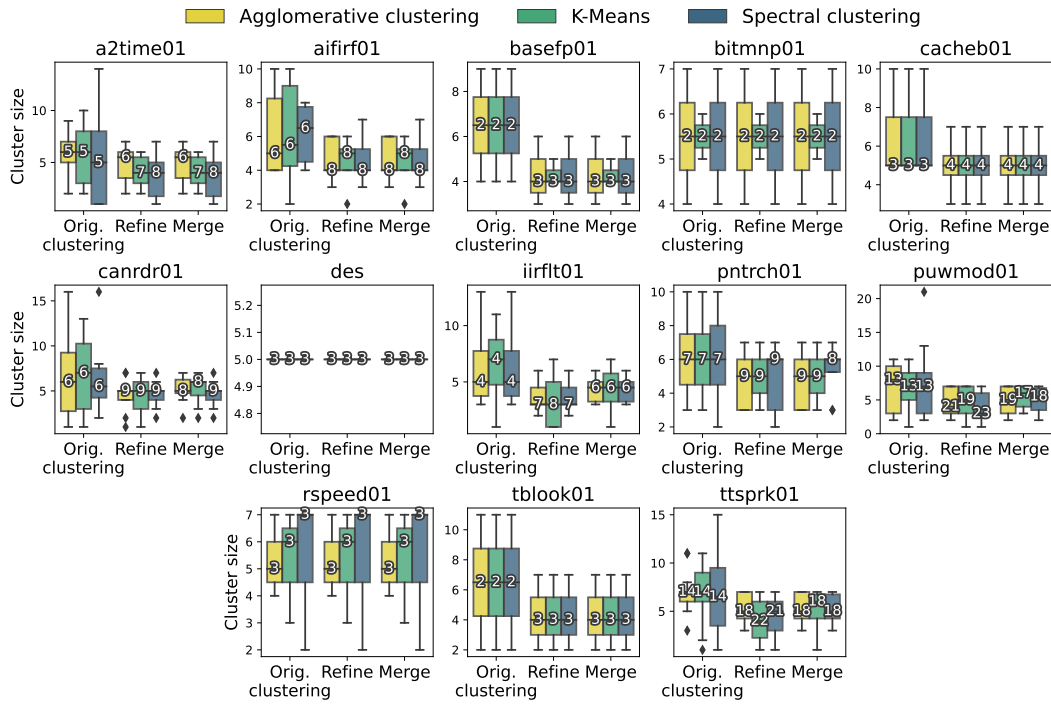


Figure 1 Cluster sizes after each stage of Algorithm 1 when using agglomerative, k-means, and spectral clustering. The numbers on the medians show the total number of clusters.

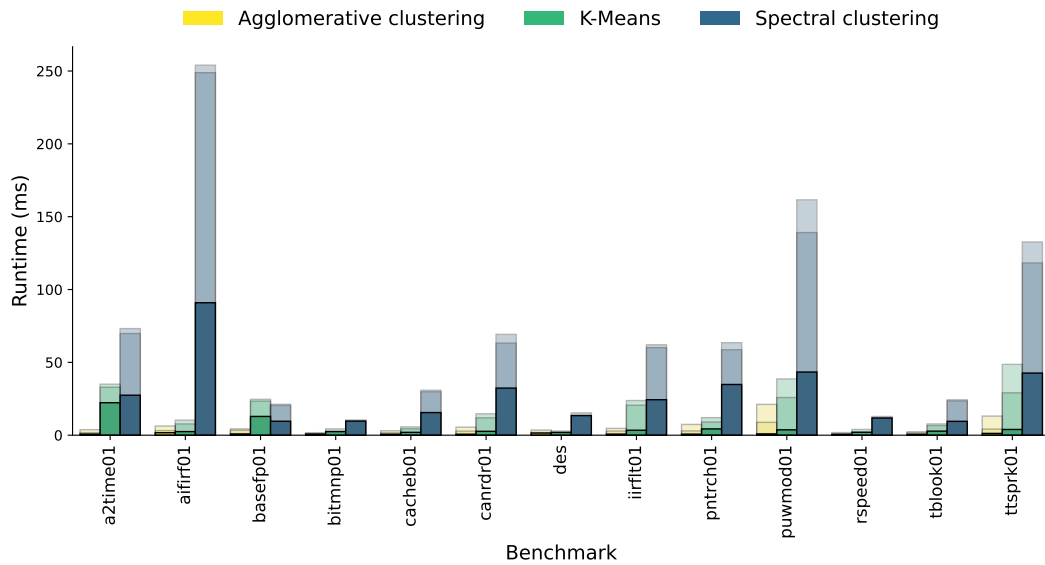
considered clustering methods. We observed the most significant reduction for benchmark `puwmod01` with spectral clustering where the merge algorithm returned 18 clusters instead of 23 produced at the refinement stage. This benchmark is one of the largest benchmarks considered in the evaluation.

► Remark 4. For large Pareto fronts, Algorithm 1 results in many clusters when the maximum cluster size τ is small. We observed these results for benchmarks `puwmod01` and `ttsprk01`. If a Pareto front is large, a system designer should iteratively invoke Algorithm 1:

1. set τ to a large value to divide the Pareto front into fewer clusters of large sizes;
 2. choose the best cluster;
 3. pass the cluster to the algorithm and decrease τ to divide the cluster into smaller clusters.
- Repeat Steps 2 and 3 until the desired maximum cluster size is achieved.

Figure 2 shows the runtime of the stages of Algorithm 1 as a stack diagram with three parts starting from 0: original clustering, refinement, and merging. For all benchmarks, the runtime of the three approaches was less than 0.3s. Agglomerative clustering resulted in the shortest runtime for all benchmarks, except `des`. For 12 out of 13 benchmarks, Algorithm 1 with spectral clustering was the slowest approach. For most benchmarks, the refinement stage was the most time-consuming part of the algorithm, whereas the merge stage was the least-time consuming part.

To sum up, in general, the three considered approaches showed very similar results according to Figure 1, but Figure 2 shows that Algorithm 1 with agglomerative clustering was finished in less than 0.03s for all benchmarks. It is 0.012s and 0.066s faster, on average, than Algorithm 1 with k-means and spectral clustering, respectively.



■ **Figure 2** Runtime of the stages of Algorithm 1 with agglomerative clustering, k-means, and spectral clustering. The parts of the stack diagram starting from 0: original clustering, refinement, and merging.

5 Conclusion

We presented a method to cluster trade-off solutions of a multiobjective optimization problem. To guarantee that the size of all clusters is less than a predefined limit, our method clusters the solutions by using a known clustering method, refines clusters with exceeding sizes, and merges small clusters if possible. The last step tries to reduce the number of clusters which may increase after the refinement.

We demonstrated our approach by clustering solutions of multiobjective function inlining problem. We compared the results by using three clustering methods as a base for our approach: k-means, agglomerative and spectral clusterings. By using the three clustering methods, the proposed clustering techniques showed similar results in terms of the number of clusters and their sizes, but it showed the smallest runtime when using agglomerative clustering.

In future work, the proposed method should be verified by using other multiobjective optimizations, including compiler-based optimizations.

References

- 1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers. <https://www.absint.com/ait/index.htm>, 2022.
- 2 Sunith Bandaru, Amos H.C. Ng, and Kalyanmoy Deb. Data mining methods for knowledge discovery in multi-objective optimization: Part B - New developments and applications. *Expert Systems with Applications*, 70:119–138, March 2017. doi:10.1016/j.eswa.2016.10.016.
- 3 Lilian Astrid Bejarano, Helbert Eduardo Espitia, and Carlos Enrique Montenegro. Clustering Analysis for the Pareto Optimal Front in Multi-Objective Optimization. *Computation*, 10(3):37, March 2022. doi:10.3390/computation10030037.
- 4 Vincenzo Catania, Giuseppe Ascia, Maurizio Palesi, Davide Patti, and Alessandro G. Di Nuovo. Fuzzy decision making in embedded system design. In *Proceedings of the 4th International*

- Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 223–228, October 2006. doi:10.1145/1176254.1176309.
- 5 Carlos A. Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer-Verlag GmbH, October 2007.
 - 6 Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, New York, NY, USA, 2001.
 - 7 Kalyanmoy Deb. Multi-objective Optimization. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 403–449. Springer US, Boston, MA, 2014. doi:10.1007/978-1-4614-6940-7_15.
 - 8 Wilm E. Donath and Albert J. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.
 - 9 Catarina Dudas, Amos H.C. Ng, and Henrik Boström. Post-analysis of multi-objective optimization solutions using decision trees. *Intelligent Data Analysis*, 19(2):259–278, April 2015. doi:10.3233/IDA-150716.
 - 10 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, July 2010. doi:10.1007/s11241-010-9101-x.
 - 11 Hisao Ishibuchi, Lie Meng Pang, and Ke Shang. Solution Subset Selection for Final Decision Making in Evolutionary Multi-Objective Optimization, June 2020. doi:10.48550/arXiv.2006.08156.
 - 12 Shashank Jadhav and Heiko Falk. Multi-Objective Optimization for the Compiler of Real-Time Systems based on Flower Pollination Algorithm. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 45–48. ACM, May 2019. doi:10.1145/3323439.3323977.
 - 13 Yanjun Kong, Yadong Mei, Xianxun Wang, and Yue Ben. Solution Selection from a Pareto Optimal Set of Multi-Objective Reservoir Operation via Clustering Operation Processes and Objective Values. *Water*, 13(8):1046, January 2021. doi:10.3390/w13081046.
 - 14 Tobias Kuhn, Carlos M. Fonseca, Luís Paquete, Stefan Ruzika, Miguel M. Duarte, and José Rui Figueira. Hypervolume Subset Selection in Two Dimensions: Formulations and Algorithms. *Evolutionary Computation*, 24(3):411–425, 2016. doi:10.1162/EVCO_a_00157.
 - 15 Congdong Li, Run Wu, and Weiming Yang. Optimization and selection of the multi-objective conceptual design scheme for considering product assembly, manufacturing and cost. *SN Applied Sciences*, 4(4):91, March 2022. doi:10.1007/s42452-022-04973-6.
 - 16 Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences – a trade-off between WCET, ACET and code size. *Software: Practice and Experience*, 41(12):1437–1458, May 2011. doi:10.1002/spe.1079.
 - 17 James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, volume 5.1, pages 281–298. University of California Press, January 1967.
 - 18 Christopher Mattson, Anoop Mullur, and Achille Messac. Minimal Representation of Multiobjective Design Space Using a Smart Pareto Filter. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Multidisciplinary Analysis Optimization Conferences. American Institute of Aeronautics and Astronautics, September 2002. doi:10.2514/6.2002-5458.
 - 19 George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956. doi:10.1037/h0043158.
 - 20 Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1998.
 - 21 Kateryna Muts. *Multiobjective Compiler-Based Optimizations for Hard Real-Time Systems*. PhD thesis, TUHH Universitätsbibliothek, December 2022. doi:10.15480/882.4799.

- 22 Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 849–856, Cambridge, MA, USA, January 2001. MIT Press.
- 23 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, January 2012. [arXiv:1201.0490v4](https://arxiv.org/abs/1201.0490v4).
- 24 Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro*, 29(5):18–29, September 2009. [doi:10.1109/MM.2009.74](https://doi.org/10.1109/MM.2009.74).
- 25 Mike Preuss and Simon Wessing. Measuring Multimodal Optimization Solution Sets with a View to Multiobjective Techniques. In *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV*, Advances in Intelligent Systems and Computing, pages 123–137, Heidelberg, 2013. Springer International Publishing. [doi:10.1007/978-3-319-01128-8_9](https://doi.org/10.1007/978-3-319-01128-8_9).
- 26 Henrik Smedberg and Sunith Bandaru. Interactive knowledge discovery and knowledge visualization for decision support in multi-objective optimization. *European Journal of Operational Research*, 306(3):1311–1329, May 2023. [doi:10.1016/j.ejor.2022.09.008](https://doi.org/10.1016/j.ejor.2022.09.008).
- 27 Abdulhamit Subasi. *Practical Machine Learning for Data Analysis Using Python*. Academic Press, London [England]; San Diego, CA, 2020.
- 28 Ling Wang, Haoqi Ni, Weifeng Zhou, Panos M. Pardalos, Jiating Fang, and Minrui Fei. MBPOA-based LQR controller and its application to the double-parallel inverted pendulum system. *Engineering Applications of Artificial Intelligence*, 36:262–268, November 2014. [doi:10.1016/j.engappai.2014.07.023](https://doi.org/10.1016/j.engappai.2014.07.023).
- 29 Xuanli L. Xie and Gerardo Beni. A validity measure for fuzzy clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):841–847, 1991. [doi:10.1109/34.85677](https://doi.org/10.1109/34.85677).
- 30 Zhongliang Yue. An extended TOPSIS for determining weights of decision makers with interval numbers. *Knowledge-Based Systems*, 24(1):146–153, February 2011. [doi:10.1016/j.knsys.2010.07.014](https://doi.org/10.1016/j.knsys.2010.07.014).

Efficient and Effective Multi-Objective Optimization for Real-Time Multi-Task Systems

Shashank Jadhav  

Hamburg University of Technology, Germany

Heiko Falk  

Hamburg University of Technology, Germany

Abstract

Embedded real-time multi-task systems must often not only comply with timing constraints but also need to meet energy requirements. However, optimizing energy consumption might lead to higher Worst-Case Execution Time (WCET), leading to an un-schedulable system, as frequently executed code can easily differ from timing-critical code. To handle such an impasse in this paper, we formulate a Metaheuristic Algorithm-based Multi-objective Optimization (MAMO) for multi-task real-time systems. But, performing multiple WCET, energy, and schedulability analyses to solve a MAMO poses a bottleneck concerning compilation times. Therefore, we propose two novel approaches – Path-based Constraint Approach (PCA) and Impact-based Constraint Approach (ICA) – to reduce the solution search space size and to cope with this problem. Evaluations showed that PCA and ICA reduced compilation times by 85.31% and 77.31%, on average, over MAMO. For all the task sets, out of all solutions found by ICA-FPA, on average, 88.89% were on the final Pareto front.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Compilers; Mathematics of computing → Discrete mathematics

Keywords and phrases Real-time systems, Multi-objective optimization, Metaheuristic algorithms, Compilers, Design space reduction

Digital Object Identifier 10.4230/OASICS.WCET.2023.5

Funding This work is part of a project that has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 779882.

1 Introduction

Modern real-time embedded systems are subject to strict constraints and must meet functional and temporal requirements, such as execution time and energy consumption. Failure to meet such constraints might lead to disastrous consequences, e.g., airbag deployment systems. For multi-task systems, schedulability is an important criterion. This paper proposes multi-objective optimization for multi-task systems that simultaneously considers WCET, energy, and schedulability for multi-task systems. The proposed framework utilizes two metaheuristic algorithms, namely Strength Pareto Evolutionary Algorithm (SPEA) [26] and Flower Pollination Algorithm (FPA) [25], to solve a static Scratchpad Memory (SPM) allocation-based MAMO problem.

A program might have different WCET- and energy-critical paths. Moreover, minimizing WCET and energy of one particular task might negatively affect others, resulting in an un-schedulable system. Furthermore, the previous single-objective optimizations treated schedulability as a constraint, which could limit design space exploration. Therefore, the proposed framework considers schedulability as an objective, which enables the identification of a Pareto front containing completely- and partially-scheduled multi-task systems. Depending on hard- and soft-real-time requirements, in the end, the system designer could choose the needed Pareto-optimal solution.



© Shashank Jadhav and Heiko Falk;

licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 5; pp. 5:1–5:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The presented MAMO approach is iterative, requires objective evaluations at every iteration, and deals with three objectives, the analysis for which can be very time-consuming. To address this issue, we introduce two novel objective-dependent approaches that generate constraints at each iteration and reduce the solution space size and the total compilation time. The first approach, PCA, uses worst- and average-case execution path information to constrain the solution space, and the second approach, ICA, uses an impact metric to constrain and reduce the solution space size. Evaluations in this paper clearly show that worst- and average-case execution information can be utilized to improve MAMO speed and solution quality. Therefore, the key contributions of this paper are:

- We formulated and solved SPM allocation-based 3-dimensional MAMO problem.
- We proposed two approaches – PCA and ICA – to reduce the solution space size.
- PCA and ICA reduced compilation times by 85.31% and 77.31%, on average, over MAMO.
- On average, 88.89% solutions found by ICA-FPA were on the final Pareto front.

This paper is outlined as follows: Sec. (2) overviews the related work. Sec. (3) defines the MAMO problem for static SPM allocation. Sec. (4) introduces the MAMO framework. Sec. (5) and (6) propose PCA and ICA, respectively. Sec. (7) presents the evaluation results. Sec. (8) present conclusions and a future work discussion.

2 Related Work

In the past, many approaches focused on SPM allocation-based single-objective WCET- or energy-aware single-task optimizations [2, 7, 12, 14, 15, 22]. Moreover, some research has considered schedulability-aware single objective optimization for multi-task systems, where schedulability is treated as a constraint [16, 17]. In contrast, this paper proposes a compiler-level multi-objective optimization for multi-task systems which can simultaneously minimize WCET, energy consumption, and schedulability objectives.

Heuristic approaches like greedy algorithms may not be optimal for multi-objective optimizations, and ILP-based approaches are well-suited for single-objective optimization. Therefore, we use metaheuristic algorithms that employ problem-independent search strategies to solve the MAMO problem. Zitzler et al. [26] introduced the SPEA algorithm, where fitness assignment is done based on the co-evolution principle. FPA, inspired by the pollination process seen in flowering plants, is a Nature-Inspired metaheuristic Algorithm (NIA) introduced by Yang et al. [24]. SPEA outperforms other EAs such as NSGA, VEGA, etc. [27], and FPA performed better than VEGA, NSGA-II, MODE, etc. [25]. Therefore, in this paper, we chose to use SPEA and FPA to solve the MAMO problem.

Evaluating every solution iteratively using metaheuristics can pose a huge bottleneck. In such scenarios, machine learning techniques can be used to predict WCET and energy objectives [10, 21]. But, their accuracy varies on hyperparameter tuning, data sample size, and the underlying machine learning model. Previously, we proposed an approximation model to approximate WCET and energy consumption of a program, which was used to perform SPM allocation [13]. This approximation model relies on the underlying compiler optimization, i.e., SPM allocation, and is not generic to any compiler-level optimization. In this paper, we propose two approaches that rely on the objectives of MAMO, specifically WCET and energy and do not rely on the underlying compiler optimization.

3 Multi-Task Multi-Objective Problem

In this section, we propose a 3-dimensional MAMO problem for a multi-task system, where we treat schedulability as an objective to minimize rather than a constraint, which allows for better solution space exploration and maintains pressure on metaheuristic algorithms during solution selection. Treating schedulability as a constraint might lead to the rejection of solutions that violate this constraint and hinder proper solution space exploration. Moreover, as the schedulability objective depends on WCET, we could consider only schedulability as a minimization objective, but it could also hinder solution space exploration. For example, a multi-task system could become schedulable, and more schedulable solutions with lower WCET values could exist within the solution space. Lastly, WCET and energy consumption objectives can contradict each other. Therefore, we simultaneously consider WCET, energy consumption, and schedulability as objectives. The proposed MAMO problem for a multi-task system is mathematically formulated as follows:

$$\begin{aligned} \min_x \quad & F(x) = (F_1(x), F_2(x), F_3(x)) \\ \text{subject to} \quad & g(x) = \sum_{t=1}^T \sum_{v=1}^{p^t} B^{tv} x^{tv} - S_{SPM} - \sum_{t=1}^T \sum_{v=1}^{p^t-1} s^{tv} |x^{tv} - x^{t_{v+1}}| \leq 0 \end{aligned} \quad (1)$$

where $x = (x^1, \dots, x^T) \in \{0, 1\}^d$ is a d -dimensional binary decision vector for a multi-task set Γ . x^t is a binary decision vector for a single task $\tau^t \in \Gamma$, where $t = \overline{1, T}$, and T is the total number of tasks. SPM allocation is a compiler optimization, where we move Basic Blocks (BB) from slow Flash to SPM. The decision of placing a BB in SPM or Flash is realized by an element $x^{tv} \in \{0, 1\}$ of the decision vector, where $v = \overline{1, p^t}$, p^t is the total number of BB s in the t^{th} task, and $d = \sum_{t=1}^T \sum_{v=1}^{p^t} v$ is the total number of BB s in the multi-task system.

$g(x) \leq 0$ is the SPM size constraint condition, where B^{tv} is the code size of BB^{tv} , BB^{tv} is the v^{th} BB in the t^{th} task, and S_{SPM} represents the SPM size. The term $|x^{tv} - x^{t_{v+1}}|$ determines if extra jump correction cost is needed, i.e., if t_v^{th} BB and the succeeding $(t_v + 1)^{\text{th}}$ BB are in different memories, then we need to perform jump correction and add the extra jump correction cost [18]. Furthermore, s^{tv} is an architecture-dependent term representing the jump correction code size. For ARM7TDMI architecture, s^{tv} is modeled as follows: $s^{tv} = 16$ if the basic block BB^{tv} ends with a jump instruction, and in case of calls, conditional jumps, or fall-through instructions $s^{tv} = 16$. Extra spill code is added if a free register is not available, which increments the jump correction cost, i.e., $s^{tv} + 4$.

$F(x) = (F_1(x), F_2(x), F_3(x))$ is the 3-dimensional objective function. $F_1(x) = \sum_{t=1}^T W^t$ and $F_2(x) = \sum_{t=1}^T E^t$ are the total WCET and energy values for the binary decision vector x , where W^t and E^t are WCET and energy consumption of task $\tau^t \in \Gamma$, respectively. $F_3(x) = \sum_{t=1}^T \rho_t W^t$ is the schedulability objective. $\rho \in \{0, 1\}^T$ is a T -dimensional binary vector, where 1's indicate that the task τ^t is a task removed from Γ , i.e., $\tau^t \in \Gamma_r$, where Γ_r denotes a set of tasks removed from Γ , such that the system of remaining tasks is schedulable. We use the ILP-based schedulability analyzer [16] to calculate the number of tasks needed to be removed from the system to achieve schedulability. Accordingly, if a system with T tasks is schedulable and all the tasks safely meet their deadlines, the analyzer will return 0. Contrary, in a worst-case scenario, the analysis will return T . MAMO returns a set of Pareto-optimal solutions that can be fully or partially schedulable and indifferent in terms of WCET and energy, which allows the system designer to choose a suitable solution from the set of Pareto-optimal binaries based on the requirements during runtime.

Algorithm 1 SPM allocation-based MAMO.

- 1: **Initialization:** Initialize the initial population, perform jump corrections, and evaluate them.
 - 2: **Input:** Initialized population and stopping criteria
 - 3: **Output:** Pareto-optimal solution set
 - 4: **while** stopping criteria is not fulfilled **do** ▷ Iterate over all generations
 - 5: **for** $j = 1 : N$ **do** ▷ Iterate over all individuals
 - 6: Update the individual using update operators
 - 7: Repair the individual if needed and perform jump correction
 - 8: Evaluate the individual
 - 9: Using the selection operator, update to next generation
-

4 Multi-objective optimization

To solve the compiler-level MAMO problem for multi-task systems, we use the WCET-aware C Compiler (WCC) [6] framework. We solve the MAMO problem using a metaheuristic algorithm and try to minimize WCET, energy consumption, and schedulability. Let $X \subset \{0, 1\}^d$ be the search space of the MAMO defined in Sec. (3). $P_i \subset X$ is the population set with N individuals at generation $i = \overline{1, M}$, where M is the maximum number of generations. $x_{i,j} \in \{0, 1\}^d$ is a d -dimensional binary individual vector at generation i , where $j = \overline{1, N}$.

Algorithm (1) presents the SPM allocation-based MAMO. The problem formulation presented in Sec. (3) is used to initialize MAMO (Line 1). After initialization, we solve MAMO by calling the metaheuristic algorithm (Lines 4-9). The metaheuristic algorithm uses an update operator to update an individual (Line 6). SPEA uses mutation and crossover [26], whereas FPA uses global and local pollination operators to update an individual [25]. The current population generation is updated after evaluations to the next using a selection operator. FPA and SPEA provide scalar fitness values to each individual and use Pareto Dominance [4] to update to the next generation. This paper considers two stopping criteria, the maximum number of generations and the maximum number of generations for which the population remains the same. After fulfilling the stopping criteria, the algorithm outputs the final Pareto-optimal solution set. The Algorithm (1) considers all the BBs in the multi-task system. The number of BBs defines the dimension of the solution space, which influences the second stopping criterion. Therefore, the smaller the solution space size, the quicker we might reach the second stopping criteria, leading to fewer individuals to evaluate. Therefore, we propose two novel approaches to reduce the solution space size in the following sections.

5 Path-based Constraint Approach

During optimization, exploring the whole solution space would be the most reliable way to find the Pareto-front. But, with limited time to perform optimization, we can consider objective-specific details to decrease the problem size. The WCET and energy objectives rely

Algorithm 2 Path-based Constraint Approach.

- 1: **Input:** Evaluated individual $x_{i,j}$
 - 2: **Output:** Constraints for next individual $x_{i+1,j}$
 - 3: Get $\mathcal{W}_{i,j}$ and $\mathcal{A}_{i,j}$, and Create $\mathcal{U}_{i,j}$.
 - 4: **for** $t = 1 : T$ **do** ▷ Iterate over all tasks
 - 5: **for** $v = 1 : p^t$ **do** ▷ Iterate over all basic blocks
 - 6: **if** $BB_{i,j}^{t,v} \notin \mathcal{U}_{i,j}$ **then**
 - 7: $x_{i+1,j}^{t,v} = 0$
-

Algorithm 3 Impact-based Constraint Approach.

```

1: Input: Evaluated individual  $x_{i,j}$ 
2: Output: Constraints for next individual  $x_{i+1,j}$ 
3: Create an empty set :  $\mathcal{H}_{i,j}$  and an empty list of  $BBs$  :  $\mathcal{B}_{i,j}$ 
4: for  $t = 1 : T$  do ▷ Iterate over all tasks
5:   for  $v = 1 : p^t$  do ▷ Iterate over all basic blocks
6:      $\mathcal{H}_{i,j} \leftarrow (BB_{i,j}^{tv}, \mathcal{M}^{tv})$ 
7: Sort ( $\mathcal{H}_{i,j}$ ) in the descending order of  $\mathcal{M}^{tv}$  values
8: for  $q = 1 : d$  do ▷ Iterate over ( $\mathcal{H}_{i,j}$ )
9:   if  $\sum_{b=1}^n B_b \leq \alpha * S_{SPM}$  then
10:     $\mathcal{B} \leftarrow \mathcal{H}_{i,j}^{q,1}$ 
11: for  $t = 1 : T$  do ▷ Iterate over all tasks
12:   for  $v = 1 : p^t$  do ▷ Iterate over all basic blocks
13:    if  $BB_{i,j}^{tv} \notin \mathcal{B}_{i,j}$  then
14:      $x_{i+1,j}^{tv} = 0$ 

```

on the Worst-Case Execution Path (WCEP) and Average-Case Execution Path (ACEP) of a program, respectively [1, 23]. WCEP and ACEP are defined as the execution paths through a task's control flow graph that leads to their WCET and Average-Case Execution Time (ACET), respectively. Therefore, instead of exploring the entire search space, we constrain the solution space at each iteration using WCEP and ACEP information. As the WCEP and ACEP of a task set could differ, PCA considers the BBs on both paths. Let $\mathcal{W}_{i,j}$ and $\mathcal{A}_{i,j}$ be the set of BBs on WCEP and ACEP of the j^{th} individual of the i^{th} generation. For every individual, BBs on WCEP and ACEP can vary. Furthermore, let $\mathcal{U}_{i,j} := \mathcal{W}_{i,j} \cup \mathcal{A}_{i,j}$. Therefore, MAMO defined by Eq. (1) is extended by adding the following constraint.

$$x_{i+1,j}^{tv} = 0, \quad \text{if } BB_{i,j}^{tv} \notin \mathcal{U}_{i,j} \quad \forall t, v \quad (2)$$

Algorithm (2) describes the PCA approach proposed in this paper. PCA takes an evaluated individual $x_{i,j}$ as an input and provides constraints for the individual $x_{i+1,j}$ from the next generation (Lines 1-2). The sets of BBs , $\mathcal{W}_{i,j}$, $\mathcal{A}_{i,j}$, and $\mathcal{U}_{i,j}$, are created (line 3). If a BB is not on WCEP or ACEP, then that element of the individual vector for the next generation is constrained to 0, and the BB is placed in Flash, i.e., $x_{i+1,j}^{tv} = 0$ (lines 4-7). This constraint is enforced by recombination and mutation operators for SPEA and local and global pollination operators for FPA. Therefore, the size and shape of the solution space can change and be differently constrained throughout the optimization run.

6 Impact-based Constraint Approach

PCA considered all BBs on the WCEP and ACEP to reduce the dimension of the solution space, but MAMO has an SPM size constraint, and SPMs are small in size. Consequently, we can assume that many BBs will not be assigned to SPM for large multi-task systems. Therefore, we propose the ICA Approach, which guides the optimization using the SPM size constraint, Worst-Case Execution Count (WCEC), and Average-Case Execution Count (ACEC) to constrain the solution space. WCEC and ACEC are defined as the number of times each basic block is executed in a worst- and average-case scenario, respectively. As the schedulability objective operates at the task level, we do not consider it for solution space reduction. While initializing ICA, we calculate the impact of each BB on the total WCET and energy when SPM allocation is not performed. We consider an individual F^* where all

the BB s are in the Flash and evaluate it by performing WCET and energy analyses. Let W_{F^*} and E_{F^*} , and $W_{F^*}^{t_v}$ and $E_{F^*}^{t_v}$ be the total WCET and total energy, and the WCET and energy of $BB_{F^*}^{t_v}$, respectively. $BB_{F^*}^{t_v}$ is the v^{th} basic block of the t^{th} task of the individual F^* . Furthermore, we calculate the impact of each BB by using $\mathbf{W}_{F^*}^{t_v} := \frac{W_{F^*}^{t_v}}{W_{F^*}}$ and $\mathbf{E}_{F^*}^{t_v} := \frac{E_{F^*}^{t_v}}{E_{F^*}}$. Let \mathcal{W}_{F^*} and \mathcal{E}_{F^*} be the set of $\mathbf{W}_{F^*}^{t_v}$ and $\mathbf{E}_{F^*}^{t_v} \forall t, v$, respectively for the individual F^* . After calculating impact values for each BB for the case where all BB s are in Flash, we call Algorithm (1) to solve the MAMO problem.

Algorithm (3) describes the proposed ICA approach used to reduce the solution space size. ICA takes an evaluated individual $x_{i,j}$ as an input and provides constraints for the individual $x_{i+1,j}$ from the next generation (Lines 1-2). ICA uses the following *impact* metric to constrain the solution space.

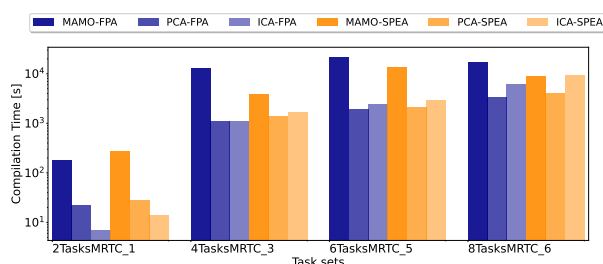
$$\mathcal{M}^{t_v} = \zeta \mathbf{W}_{F^*}^{t_v} * w_{i,j}^{t_v} + \beta \mathbf{E}_{F^*}^{t_v} * a_{i,j}^{t_v}, \text{ where } \mathbf{W}_{F^*}^{t_v} \in \mathcal{W}_{F^*} \ \& \ \mathbf{E}_{F^*}^{t_v} \in \mathcal{E}_{F^*} \quad (3)$$

The terms $w_{i,j}^{t_v}$ and $a_{i,j}^{t_v}$ represent the WCEC and ACEC of $BB_{i,j}^{t_v}$ of the individual j at generation i , respectively. Furthermore, $\zeta, \beta \in [0, 1]$ and $\zeta + \beta = 1$ are the positive constant weights assigned to WCET and energy terms. We can use ζ and β to adjust the weight of the objectives. During SPM allocation, the BB s on WCEP and ACEP affect the program's total WCET and energy. But, BB s having higher WCEC and ACEC values does not indicate that the BB will have higher WCET and energy values. Therefore, the WCEC and ACEC term in Eq. (3) is multiplied by the WCET and energy impact terms to calculate the BB 's impact. The bigger the value of \mathcal{M}^{t_v} , the higher the overall impact of $BB_{i,j}^{t_v}$ on the total WCET and energy of the individual.

An empty set $\mathcal{H}_{i,j}$ of the ordered pairs of BB and *impact* metric is initialized (line 3), i.e., $(BB_{i,j}^{t_v}, \mathcal{M}^{t_v}) \in \mathcal{H}_{i,j} \forall t, v$. Furthermore, an empty list $\mathcal{B}_{i,j}$, which will contain the list of BB s selected by this heuristic, is initialized (line 3). We iterate through the multi-task system, i.e., $d = \sum_{t=1}^T \sum_{v=1}^{p^t} v$ times, to fill the set $\mathcal{H}_{i,j}$ and sort this set in the descending order of \mathcal{M}^{t_v} values (lines 4-7). For selecting the BB s, we iterate through $\mathcal{H}_{i,j}$ and add BB s to $\mathcal{B}_{i,j}$ until the total size of the BB s selected, i.e., $\sum_{b=1}^{\eta} B_b$, is not greater than $\alpha * S_{SPM}$, where α is a positive constant, and S_{SPM} is the SPM size (Lines 8-10). Furthermore, B_b is the size of the b^{th} BB in $\mathcal{B}_{i,j}$ and $\eta \in \mathbb{N}$ such that $1 \leq \eta \leq d$ represents the total number of BB s which are not constrained for the $(i+1)^{\text{th}}$ generation. Using this heuristic, we select the BB s that have the highest impact on the WCET and energy of the code. We further constrain the solution space by limiting the number of the BB s selected by $\alpha * S_{SPM}$. If a BB is not in $\mathcal{B}_{i,j}$, then that element of the individual for the next generation is constrained to 0, i.e., $x_{i+1,j}^{t_v} = 0$ (Lines 11-14). This constraint is enforced by SPEA's recombination and mutation operators and FPA's local and global pollination operators to update the individual to $(i+1)^{\text{th}}$ generation. Therefore, if the BB is not in $\mathcal{B}_{i,j}$, i.e., $BB_{i,j}^{t_v} \notin \mathcal{B}_{i,j}$, then $x_{i+1,j}^{t_v}$ BB of individual j will be placed in Flash during the $(i+1)^{\text{th}}$ generation. Using this heuristic, we drastically constrain the solution space of big multi-task problems and perform MAMO within a limited timeframe. As ICA considers WCEC and ACEC information, it relies indirectly on the properties considered in PCA. But, unlike PCA, the ICA does not ignore the BB s not on WCEP and ACEP, but they have a low priority.

7 Evaluation

In this section, the evaluations compare the MAMO approach with PCA and ICA. The evaluations include compilation times, Pareto fronts, and quality indicators comparison for task sets consisting of 2, 4, 6, and 8 tasks. For each, 10 task sets are randomly generated



■ **Figure 1** Compilation times required to perform MAMO, PCA, and ICA.

and are un-schedulable by construction. Multiple single-task benchmarks from the MRTC suite [9], with loop bound annotations from the TACLeBench [5], were combined into a multi-tasking task set. Optimization flag `-O2`, which enables several ACET-oriented compiler optimizations, was used, and SPM size was set to 60% of the code size of each task set to increase the pressure on MAMO. A timeout value of 200 h is set for the optimization. If the optimization does not finish within this time limit, the final Pareto front is generated from the last generation, and the final results are output. The SPM allocation code generated by WCC is for ARM7TDMI architecture. For the sake of brevity, the figures show results for four randomly chosen task sets¹. WCC uses an external WCET analyzer called *aiT* [1] for WCET analysis and an internal schedulability analyzer [16] for schedulability analysis. Furthermore, WCC uses a cycle-true instruction set simulator from Synopsys called Virtualizer [11] to get ACEC and ACET data. The energy analyzer within WCC uses the energy model proposed by Roth et al. [20] and average-case data to perform the analysis [23].

For the FPA-based optimization, the switch probability between the local and global pollination is $p_s = 0.8$. According to [25, 19], for FPA, the positive integer $\lambda = 1.5$ for the standard gamma function, and the scaling factor $\gamma = 0.1$ works well. For SPEA-based optimization, the external population set size is 10. The crossover probability is 0.8, and the mutation probability is 0.2. The population size is 10, the first stopping criterion –the maximum number of generations– is 80, and the second stopping criterion –the maximum number of generations for which the population remains the same– is 10. PCA does not need any extra parameter settings. But, for ICA, we set the weights for \mathcal{M} as $\zeta = 0.5$ and $\beta = 0.5$ so that the WCET and energy objectives are equally weighted while selecting the *BBs*. Furthermore, $\alpha = 0.9$, i.e., we allow the *BBs* 0.9 times the SPM size during selection. α is less than 1 to accommodate the extra code inserted during jump correction. The results from these evaluations are valid for the algorithm parameters described above.

7.1 Compilation Times

Fig. (1) compares the compilation times for MAMO, PCA, and ICA on four task sets. The x - and y -axis represent task sets and compilation times, respectively. Each task set has six bars – the first three are the results for FPA, and the last three for SPEA, respectively, for MAMO, PCA, and ICA. The figure shows that MAMO took more time to output the final Pareto front than PCA and ICA. Overall evaluations show that PCA and ICA achieved 85.31% and 77.31% overall reduction in compilation times, respectively, compared to MAMO. Moreover, PCA–FPA achieved the most reduction in compilation time. The timeout value is hit by

¹ All the remaining figures can be made available at the readers' request.

17, 6, and 7 task sets in the case of MAMO, PCA, and ICA, respectively. MAMO-FPA, on average, required 184.41% higher compilation time than MAMO-SPEA. PCA-FPA and ICA-FPA required 26.8% and 46.99% less compilation times than PCA-SPEA and ICA-SPEA, respectively. The improved performance of FPA over SPEA in PCA and ICA may be due to constraints on the search space, which make FPA update strategies for PCA and ICA more effective in converging to the Pareto-optimal front.

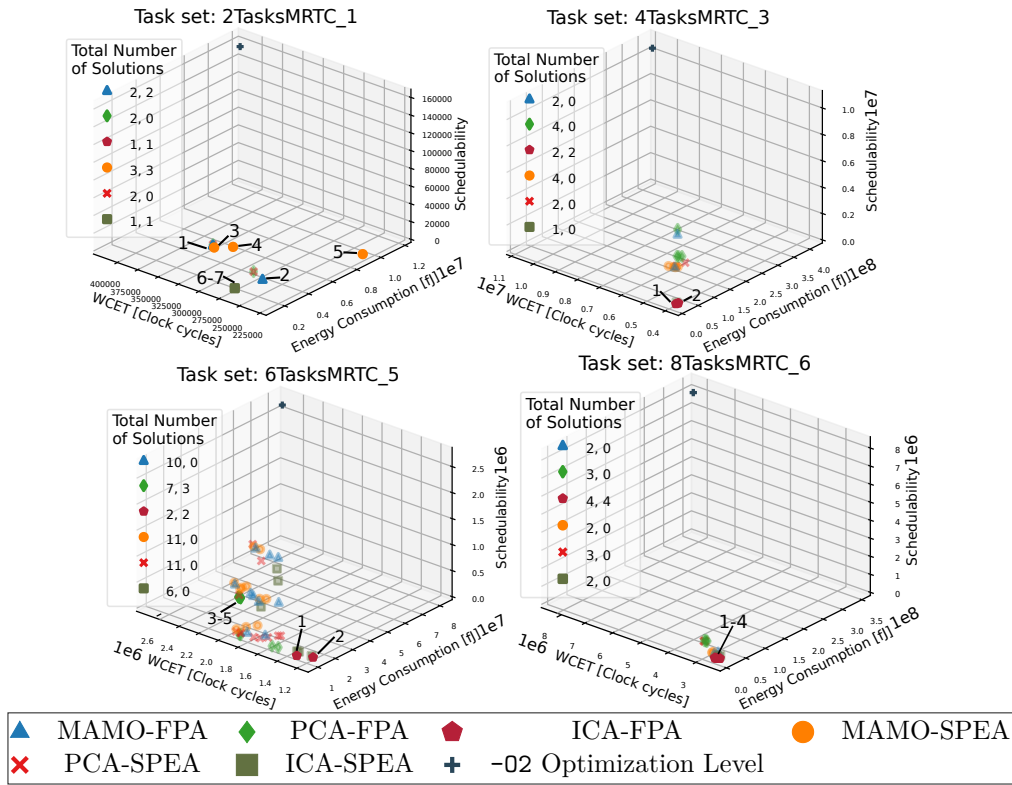
The overall decrease in compilation times achieved using PCA and ICA is due to the search space reduction and the second stopping criterion. The total number of *BBs* in the task set determines the search space size. To calculate the reduction in the search space for PCA, we first calculated the total number of *BBs* not constrained by Eq. (2). As the number of *BBs* on WCEP and ACEP may vary, we calculated the average number of *BBs* on WCEP and ACEP over all populations. Similarly, for ICA, the number of *BBs* not constrained according to the Algorithm (3) are calculated and averaged over all the populations. PCA and ICA, on average, achieved 60.06% and 87.638% reduction in the search space, respectively. The reduction in the search space can cause the optimization to hit the second stopping criteria faster. But, the average reduction in the search space cannot directly reflect the decrease in compilation times. If the metaheuristics find better solutions in every generation, the second stopping criteria is not fulfilled. E.g., even though ICA achieved a higher reduction in the search space size, PCA-SPEA achieved the most reduction in compilation time on average.

7.2 Pareto Fronts

Obtaining a true Pareto front to MAMO problem described in Sec. (3) is ambitious. Under a realistic assumption that the true Pareto front is unknown, we define a new set \mathcal{P} that represents its approximation. Let A and B be two Pareto fronts returned by FPA and SPEA, respectively. The new set \mathcal{P} is defined as the set of all non-dominated points of the union of the sets A and B , i.e., $\mathcal{P} = \{p_i | \forall p_j \in (A \cup B) \prec p_i\}$. For each task set, we obtain the final Pareto front (\mathcal{P}) from the union of all the Pareto optimal solutions obtained from all the approaches. Fig. (2) shows the solutions for four task sets found by MAMO, PCA, ICA, and the standard -02 optimization level in the form of Pareto fronts. We represent WCET, energy, and schedulability values on the x -, y -, and z -axis, respectively. In the legend of the sub-figure, we show the number of solutions returned by each approach. Furthermore, we show the number of solutions on \mathcal{P} out of the total solutions. To distinguish the solutions on \mathcal{P} , they are highlighted using numbers.

For the task set 2TasksMRTC_1, we see that all solutions by MAMO and ICA are on \mathcal{P} . One MAMO-FPA solution (Solution 2) is scheduled, whereas the other is schedulable if one task is removed from the task set. Similarly, one MAMO-SPEA solution out of three (Solution 5) has a schedulable task set. ICA-FPA and ICA-SPEA found one completely scheduled solution each (Solution 6 and 7) on \mathcal{P} , converging on the same solution. None of the solutions obtained by PCA are on \mathcal{P} . Considering the compilation times by ICA and MAMO, we can conclude that ICA performed better for this task set. Besides, when we compare the solutions on \mathcal{P} with the -02 solution, we see on average 34.02%, 78.69%, and 91.781% decrease in WCET, energy consumption, and schedulability objectives, respectively. The 91.781% average decrease in the schedulability objective indicates that \mathcal{P} consists of partially scheduled solutions too.

For task set 4TasksMRTC_3, ICA-FPA clearly outperforms others. Both ICA-FPA's solutions on \mathcal{P} are completely schedulable. For the task set 6TasksMRTC_5, PCA-FPA and ICA-FPA found three and two solutions on \mathcal{P} , respectively. ICA-FPA solutions are



■ **Figure 2** Solutions Obtained by MAMO, PCA, ICA, and -O2 runs for multi-task system.

completely schedulable, and PCA-FPA solutions are schedulable if one task is removed from the task set. But, PCA-FPA found these solutions faster than ICA-FPA (cf. Fig. (1)). For task set 8TasksMRTC_6, ICA-FPA was able to find better solutions overall. We can see that the time taken by PCA-FPA is lower than ICA-FPA (cf. Fig. (1)). But, the quality of the solutions obtained by ICA is much better than PCA. Consequently, we can say that ICA-FPA performs better for this task set. Furthermore, when we compare the solutions on \mathcal{P} with the -O2 solution, we see on average 68.23%, 97.09%, and 100% decrease in WCET, energy consumption, and schedulability objectives, respectively. The 100% decrease in the schedulability objective indicates that all the solutions on \mathcal{P} are completely schedulable.

From overall evaluations, for all the task sets, out of all solutions found by MAMO for FPA and SPEA, on average, 10.24% and 13.39% of solutions were on \mathcal{P} . PCA for FPA and SPEA had, on average, 11.27% and 5.93% of solutions on \mathcal{P} . Finally, ICA for FPA and SPEA algorithms had 88.89% and 4.81% of solutions on \mathcal{P} , respectively, on average. Therefore, we can say that ICA-FPA provided most solutions on \mathcal{P} .

7.3 Quality Metrics

We use the following quality metrics to evaluate and compare the quality of multi-objective optimization approaches. *Coverage* (C_A) [26] is a quality metric that describes the total number of dominated points in a set A . The lower the value of C_A , the better. The *Non-Dominance Ratio* (NDR_A) [8] is another quality metric measuring the ratio of non-dominated solutions contributed by a particular solution set A to the non-dominated solutions provided by all solution sets. The higher the value of the non-dominance ratio, the better. The

Non-Dominated Solutions (NDS_A) [3] is the last considered quality metric that calculates the number of non-dominated solutions concerning the set A itself, when compared to \mathcal{P} . The higher the value of non-dominated solutions, the better.

■ **Table 1** Comparing FPA & SPEA.

■ **Table 2** Comparing MAMO, PCA, & ICA.

Quality metric	Number of Task Sets					
	MAMO		PCA		ICA	
	FPA	SPEA	FPA	SPEA	FPA	SPEA
C_A	21	19	23	17	37(3)	0(3)
NDR_A	19(2)	19(2)	20(3)	17(3)	40	0
NDS_A	21	19	23	17	37(3)	0(3)

Quality metric	Number of Task Sets					
	FPA			SPEA		
	MAMO	PCA	ICA	MAMO	PCA	ICA
C_A	3(4)	2(4)	27(3)	3(3)	1(3)	0(3)
NDR_A	3(4)	2(4)	27(3)	3(3)	1(3)	0(3)
NDS_A	3(4)	2(4)	27(3)	3(3)	1(3)	0(3)

We compared quality metrics to evaluate the quality of the obtained solutions. We first compared solutions obtained using FPA and SPEA algorithms for each approach. For this comparison, \mathcal{P} is generated individually for MAMO, PCA, and ICA by combining the final Pareto fronts of their respective FPA and SPEA runs. Table (1) provides the total number of task sets for which FPA and SPEA performed better for each approach. During evaluations, we encountered task sets for which the quality metrics were indifferent to each other. The total number of such indifferent task sets is indicated using brackets within the table. From an overall comparison, we can say that SPEA and FPA algorithms provided relatively the same quality of solutions for MAMO and PCA, and FPA performed outright better for ICA.

Furthermore, we compared MAMO, PCA, and ICA approaches in terms of the above-mentioned quality metrics. For this comparison, \mathcal{P} is generated by combining the final Pareto fronts of all the approaches. Table (2) provides the total number of task sets for which MAMO, PCA, and ICA performed better. The total number of task sets with indifferent quality metrics are indicated within the table using brackets. From an overall comparison, we can clearly see that ICA–FPA provided better quality solutions for most task sets. Although it is difficult to know beforehand which approach will deliver the better Pareto-optimal solutions due to the non-deterministic nature of algorithms, we could find better results by constraining the solution space and focusing the optimization direction on the BBs that highly impact the objectives.

8 Conclusions

In this paper, we formulated a 3-dimensional SPM allocation-based MAMO and solved it using FPA and SPEA algorithms. As the compilation times required for the optimization can increase with the problem size, we introduced two new approaches, PCA and ICA, to cope with the MAMO problem size. All these approaches were able to find the trade-offs between schedulability, WCET, and energy consumption. Moreover, we compared the results obtained using FPA and SPEA for all three approaches. From evaluations, we, on average, achieved 85.31% and 77.31% reduction in compilation times using PCA and ICA compared to MAMO, respectively. Moreover, ICA–FPA found good quality solutions for 27 task sets and, on average, found 88.89% of the solutions on \mathcal{P} , which is the highest compared to other considered optimization approaches. Therefore, in this paper, we were able to show that clever integration of worst-case and average-case information within the optimization can lead to a drastic reduction in compilation times and help find better-quality solutions.

The two approaches discussed in this paper to reduce the solution space effectively reduced the runtime of the optimization and provided quality solutions. But, there were still some larger task sets that ran into timeouts during our evaluations. In the future, a

promising approach to further reduce the compilation times could be to partially replace time-consuming WCET and energy analyses with pessimistic WCET and energy estimations. In this paper, the approximated Pareto-optimal solution set returned by the optimization could consist of either fully or partially schedulable solutions. But a multi-task system could consist of tasks with both hard and soft timing constraints. Therefore, in the future, we could extend the proposed multi-objective formulation in which the system designer specifies tasks with hard and soft timing constraints. Based on these specifications, the compiler could either treat a task as part of the schedulability constraint or part of the schedulability objective, which could further constrain the solution space and provide the system designer more control over the optimization parameters and the compiler output. Furthermore, SPM allocation is just one optimization that we have considered as a multi-task multi-objective problem. Other compiler-based optimizations also have great potential, which might need objective-independent ways to constrain the solutions space. Therefore, in the future, we can consider hybrid algorithms, which combine relaxed ILPs and metaheuristic algorithms together to constrain the solution space in a problem-independent manner.

References



- 1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2021.
- 2 Anuradha Balasundaram and Vivekanandan Chenniappan. Optimal code layout for reducing energy consumption in embedded systems. In *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*, pages 1–5. IEEE, 2015.
- 3 Sanghamitra Bandyopadhyay and Arpan Mukherjee. An algorithm for many-objective optimization with reduced objective computations: A study in differential evolution. *IEEE Transactions on Evolutionary Computation*, 19(3):400–413, 2014.
- 4 Michael TM Emmerich and André H Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural computing*, 17:585–609, 2018.
- 5 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.
- 6 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.
- 7 Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 143–148, 2007. DOI 10.1145/1289816.1289853.
- 8 Chi-Keong Goh and Kay Chen Tan. A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 13(1):103–127, 2008.
- 9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The malmödalén wcet benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
- 10 Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A new hybrid approach on wcet analysis for real-time systems using machine learning. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- 11 Synopsys Inc. Comet system engineering ide, Online. URL: <http://www.synopsys.com>.

- 12 Yuriko Ishitobi, Tohru Ishihara, and Hiroto Yasuura. Code placement for reducing the energy consumption of embedded processors with scratchpad and cache memories. In *2007 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 13–18. IEEE, 2007.
- 13 Shashank Jadhav and Heiko Falk. Approximating wcet and energy consumption for fast multi-objective memory allocation. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 162–172, 2022.
- 14 Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 612–617, 2006.
- 15 Yooseong Kim, David Broman, and Aviral Shrivastava. WCET-Aware Function-Level Dynamic Code Management on Scratchpad Memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):1–26, 2017.
- 16 Arno Luppold and Heiko Falk. Schedulability aware wcet-optimization of periodic preemptive hard real-time multitasking systems. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 101–104, 2015.
- 17 Arno Luppold and Heiko Falk. Schedulability-aware spm allocation for preemptive hard real-time systems with arbitrary activation patterns. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1074–1079. IEEE, 2017.
- 18 Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical challenges of ilp-based spm allocation optimizations. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, pages 86–89. ACM, 2016.
- 19 Douglas Rodrigues, Xin-She Yang, André Nunes De Souza, and João Paulo Papa. Binary flower pollination algorithm and its application to feature selection. In *Recent advances in swarm intelligence and evolutionary computation*, pages 85–100. Springer, 2015.
- 20 Mikko Roth, Arno Luppold, and Heiko Falk. Measuring and modeling energy consumption of embedded systems for optimizing compilers. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 86–89. ACM, 2018.
- 21 Akash Sachan and Bibhas Ghoshal. Learning based compilation of embedded applications targeting minimal energy consumption. *Journal of Systems Architecture*, 116:102116, 2021.
- 22 Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation & Test in Europe (DATE)*, pages 409–415, 2002.
- 23 TeamPlay Consortium. Deliverable D3.1 - Report on the TeamPlay Basic Compiler Infrastructure - Version 1.0, 2018.
- 24 Xin-She Yang. Flower pollination algorithm for global optimization. In *International conference on unconventional computing and natural computation*, pages 240–249. Springer, 2012.
- 25 Xin-She Yang, Mehmet Karamanoglu, and Xingshi He. Flower pollination algorithm: a novel approach for multiobjective optimization. *Engineering Optimization*, 46(9):1222–1237, 2014.
- 26 Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*, volume 63. Citeseer, 1999.
- 27 Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

Towards Multi-Objective Dynamic SPM Allocation

Shashank Jadhav  

Hamburg University of Technology, Germany

Heiko Falk  

Hamburg University of Technology, Germany

Abstract

Most real-time embedded systems are required to fulfill timing constraints while adhering to a limited energy budget. Small ScratchPad Memory (SPM) poses a common hardware constraint on embedded systems. Static SPM allocation techniques are limited by the SPM's stringent size constraint, which is why this paper proposes a Dynamic SPM Allocation (DSA) model at the compiler level for the dynamic allocation of a program to SPM during runtime. To minimize Worst-Case Execution Time (WCET) and energy objectives, we propose a multi-objective DSA-based optimization. Static SPM allocations might inherently use SPM sub-optimally, while all proposed DSA optimizations are only single-objective. Therefore, this paper is the first step towards a DSA that trades WCET and energy objectives simultaneously. Even with extra DSA overheads, our approach provides better quality solutions than the state-of-the-art multi-objective static SPM allocation and ILP-based single-objective DSA approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Compilers; Mathematics of computing → Discrete mathematics

Keywords and phrases Multi-objective optimization, Embedded systems, Compilers, Dynamic SPM allocation, Metaheuristic algorithms

Digital Object Identifier 10.4230/OASICS.WCET.2023.6

Funding This work is part of a project that received funding from NXP Semiconductors.

1 Introduction

Real-time embedded systems must satisfy hard timing constraints and often operate on a limited energy budget. To optimize such systems, it is important to consider WCET and energy consumption of the program. As SPMs are fast and energy-efficient local memories, various static SPM allocation-based optimizations have been explored to exploit their potential. But, their small size gravely constrains the static optimization problem. Therefore, we propose a compiler-level DSA model in this paper to exploit the memory subsystem and circumvent the SPM size constraint. Additionally, for the very first time, we propose a strategy to perform WCET and energy analyses of such dynamically allocated programs statically at compile-time, enabling us to perform DSA-based multi-objective optimization during compilation.

DSA is traditionally an important task for Operating Systems (OS), but the execution times of OS-based allocation techniques are difficult to predict and guarantee. The compiler-based DSA has been investigated before for reasonably limited architectures, and only single-objective optimizations to minimize either WCET or energy have been considered. However, the program's WCET- and energy-critical areas may differ, and optimizing for WCET alone can negatively impact energy consumption and vice versa. Therefore, in this paper, we propose for the very first time a multi-objective optimization that uses the proposed DSA model and simultaneously optimizes the WCET and energy consumption.

We implemented the proposed DSA-based multi-objective optimization within the WCET-aware C Compiler (WCC) [6] framework and solved using two metaheuristic algorithms, namely Flower Pollination Algorithm (FPA) [25] and Strength Pareto Evolutionary Algorithm



© Shashank Jadhav and Heiko Falk;

licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 6; pp. 6:1–6:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(SPEA) [26]. For the sake of brevity, we refer to this optimization run as MO_D in this paper. Furthermore, we compare the evaluation results of MO_D with a static SPM allocation-based multi-objective optimization and an ILP-based single-objective DSA optimization referred to as MO_S and SO_D , respectively, in this paper. The key contributions of this paper are:

- For the very first time, we formulated a DSA model that allocates memory objects from Flash to SPM during runtime and supports both WCET and energy analyses.
- We proposed a MO_D that uses metaheuristic algorithms to solve the said problem.
- MO_D is compared with MO_S and SO_D using real-world benchmark suites from EEMBC.
- DSA introduces significant WCET and energy overheads still, the quality of solutions from MO_D is slightly better than the ones from MO_S , and MO_D outperforms SO_D .

The paper is organized as follows: Sec. (2) provides an overview of the related work. Sec. (3) discusses the proposed DSA model. Sec. (4) presents MO_D , and Sec. (5) presents the evaluation results. A conclusion and discussion of the future work conclude the paper.

2 Related Work

DSA facilitates copying code and data objects on and off memories during runtime and exploits the memory subsystem to its fullest potential [13]. DSA-based approaches proposed in the past are in the context of single-objective optimizations [22, 19, 7, 20]. Deverge et al. [5] proposed DSA for static and stack data to minimize WCET. Kim et al. [14] proposed dynamic instructions allocation at the function level for minimizing WCET using direct memory access transfers. Verma et al. [24] proposed an SPM overlay approach for data and instruction allocation that minimizes the energy consumption of the program. Liu et al. [15] built upon Verma’s scratchpad overlay model by considering a multi-level SPM architecture for multi-core processors for minimizing WCET. These approaches consider the DSA of either code or data and focus on only single objective optimization.

Performing compiler-level multi-objective optimizations has rarely been exploited. Lokuciejewski et al. [16] proposed a stochastic evolutionary approach to find Pareto optimal compiler optimization sequences. He considered trade-offs between Average-Case Execution Time (ACET) and WCET, as well as WCET and code size. Muts et al. [17] proposed a function-inlining-based multi-criteria optimization that traded WCET, energy, and code size. Hoste et al. [10] proposed a multi-objective optimization framework that used evolutionary algorithms to explore compiler optimization levels and automatically finds Pareto-optimal optimization levels. In the past, we proposed a multi-objective optimization using FPA to perform compiler-level static SPM allocation [11]. However, none of these multi-objective optimizations focus on dynamic allocation to exploit the memory subsystems. Therefore, this paper is the first step toward performing DSA-based multi-objective optimization that simultaneously minimizes the WCET and energy consumption of the program.

3 Dynamic SPM Allocation Model

In this section, we propose a DSA model within WCC that can dynamically allocate memory objects at runtime. DSA is the process of allocating memory objects dynamically during the runtime of a program. Performing compiler-level DSA allows us to predetermine the WCET- and energy-intensive memory object that could be dynamically copied to SPM during runtime such that WCET and energy objectives are minimized.

3.1 Memory Objects

A memory object (*memObj*) is defined as the finest granularity program fragment considered in a DSA problem. A Basic Block (BB), a code sequence with no branches except possibly at the exit, can be considered a *memObj*. But, while performing DSA, the re-usability of a *memObj* plays a critical role. Executing a *memObj* only once from SPM leads to certain WCET or energy reductions, but the overheads for dynamically copying such *memObj* can overshadow these savings. For this reason, this paper considers only *memObjs* that are executed several times, i.e., that exhibit a high re-usability. By construction, such *memObjs* are loops and functions. Therefore, we consider functions and loops as code memory objects (*memObj_c*) for dynamic allocation. Loops and functions can provide the re-usability of BBs and reduce additional overhead introduced by the movement of individual BBs.

A global data variable is *live* throughout the complete execution of the code. The dynamic allocation of global data variables can introduce unnecessary overheads in terms of WCET and energy consumption. Therefore, global data variables (*memObj_d*) are allocated statically either to SPM or to Flash by our approach. On the other hand, the scope of local data variables only exists within some parts of a single function. Therefore, our approach considers local data variables as part of the functions or loops within which they are being used and are dynamically allocated in conjunction with them.

The underlying exemplary architecture considered while modeling the DSA model consists of the Flash, an instruction SPM (*ISPM*), and a data SPM (*DSPM*). Let $\mathcal{M} \subset \mathcal{F} \cup \mathcal{L} \cup \mathcal{G}$ be a set of *memObj* that is a union of the set of functions \mathcal{F} , the set of loops \mathcal{L} , and the set of global data variables \mathcal{G} within a program. Let $x \in \{0, 1\}^d$ represent a d -dimensional binary decision variable vector that describes which *memObj* is allocated in which memory. x is a block vector, where the subvector $x_{1:F} = (x_1, \dots, x_F)$ are decision variables for functions, the subvector $x_{(F+1):(F+L)} = (x_{F+1}, \dots, x_{F+L})$ are decision variables for loops, and the subvector $x_{(F+L+1):(F+L+G)} = (x_{F+L+1}, \dots, x_{F+L+G})$ are decision variables for global data variables. F is the total number of functions, L is the total number of loops, and G is the total number of global data variables within \mathcal{M} . $d = (F + L + G)$ is the total number of *memObj*. Each coordinate $x_i, i = \overline{1, d}$ of vector x corresponds to a specific *memObj*:

$$x_i = \begin{cases} 1, & \text{if } \mathcal{M}_i \text{ is in } ISPM \\ 0, & \text{if } \mathcal{M}_i \text{ is in Flash} \end{cases} \quad x_i = \begin{cases} 1, & \text{if } \mathcal{M}_i \text{ is in } DSPM \\ 0, & \text{if } \mathcal{M}_i \text{ is in Flash} \end{cases} \quad (1)$$

$$\forall i = \overline{1, (F + L)} \quad \forall i = \overline{(F + L + 1), (F + L + G)}$$

The *memObj_c* referring to decision variables $x_i, \forall i = \overline{1, (F + L)}$ are allocated dynamically from Flash to *ISPM*, and the *memObj_d* referring to $x_i, \forall (F + L + 1), (F + L + G)$ are allocated statically from Flash to *DSPM*.

3.2 Liveness Analysis

A *memObj* is *live* at an edge $e \in E$ of the control flow graph $G(N, E)$ if there exists a back path from the edge e to a node $n \in N$, where the *memObj* is defined without being redefined at any other node along the path [24]. For the sake of brevity, a detailed explanation of the standard liveness analysis is omitted [2]. We perform liveness analysis within WCC at the function level. Each function is analyzed to determine the *live* range of a *memObj*. A function *memObj* is defined (*def*) when it is first called within the function currently under analysis. The subsequent calls for that function are categorized as *use*. The *live* range of the function *memObj* is the path, i.e., set of BBs, from *def* until the last *use*.

A loop *memObj* is defined at the live-out edge of the loop-entry BB. All BBs within the loop are categorized as *use*. The *live* range of the loop *memObj* is the loop-entry's predecessor BB, i.e., *def*, until the loop-exit BBs. In case of multiple entries and exits for a loop, all the BBs are considered part of the *live* range of the loop *memObj*. In the case of nested loops, each loop is considered an individual *memObj* entity. The loop *memObj* nested within another loop is defined at the live-out edge of the top-enclosing loop entry's BB, and the BBs within the individual loop *memObj* are categorized as *use*. The *live* range of a nested loop *memObj* spans over all the BBs contained within the top-enclosing loop. In the case of global data variables, they are considered *live* throughout the complete execution of the code.

Let $\Lambda = \{\lambda_1, \dots, \lambda_{(F+L)}\}$, where $(F+L)$ is the total number of *memObj_c* and λ_i be a set of BBs *live* for *memObj_c* M_i , i.e., $\lambda_i = \{b_p \mid \forall p \ b_p \text{ is live for } M_i\}$. Let $C = \{c_1, \dots, c_{(F+L)}\}$, where c_i be a $(F+L)$ -dimensional binary vector that determines if there exists an overlap of *live* ranges between the i^{th} *memObj_c* and others. Each coordinate $c_{ij}, j = \overline{1, (F+L)}$ corresponds to a conflict of i^{th} *memObj* with the j^{th} *memObj*, i.e.,

$$c_{ij} = \begin{cases} 1, & \text{if } \exists b_p \mid b_p \in \lambda_i \ \& \ b_p \in \lambda_j, \ \& \ i \neq j \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

When solving the address assignment problem, *liveness* conflicts between *memObjs* are considered. These conflicts prevent the allocation of *memObjs* that share conflicting *liveness* to the same memory address within SPM.

3.3 Address Assignment

DSA allows copying of *memObj_c* from Flash to SPM during runtime dynamically. But, we must be careful that any *memObj_c* is not overwritten during its execution. Therefore, we need to solve an address assignment problem at compile-time such that no two *memObj_c* that are *live* at the same time are allocated to the same memory addresses. We solve the address assignment algorithm within WCC to appropriately allocate address spaces to *memObj_c* for proper dynamic allocation. Moreover, all BBs contained within a *memObj_c* are not always placed within consecutive memory addresses. For example, a loop within a nested loop could have BBs located in consecutive memory addresses, followed by another loop, and then followed by the remaining BBs. In this case, we need two distinct memory copy functions to dynamically allocate the whole *memObj*. Therefore, we define an address object (*addrObj*) as a set of BBs from the *memObj_c* such that they are placed within consecutive memory addresses. Solving the address assignment problem provides us with the start address, the destination address, and the size of the *addrObj*, which are needed for their dynamic allocation during runtime. Let T_i be the total number of *addrObjs* associated with the *memObj_c* \mathcal{M}_i , and \mathcal{T} be the total number of *addrObjs* that need dynamic allocation.

To solve the address assignment problem, we use a combination of the first-fit and best-fit heuristics [8]. The first-fit heuristic fit as many *addrObj* as possible within SPM until the SPM is full. If two *addrObj* are adjacent within the Flash, then we try to place them similarly in SPM. Once the SPM is full, we run the best-fit heuristic to find the best possible place in SPM for the remaining *addrObj*. *memObjs* with *liveness* conflicts are not overlapped within the memory addresses. In case the size of the *addrObj* is larger than the already placed *addrObjs*, then we try to find multiple adjacent *addrObj* that fit the considered *addrObj*. If all *addrObjs* are assigned to SPM, then the algorithm returns 0. If that is not the case, then it returns $(\mathcal{T} - \eta)$, where η is the number of *addrObjs* assigned to SPM.

Algorithm 1 Generation and Analyses of Dynamically Allocated Code.

```

1: For each solution  $x$ 
2: for  $i = 1 : (F + L + G)$  do ▷ For each  $\mathcal{M}_i$ 
3:   if  $x_i == 1$  then ▷ If  $\mathcal{M}_i$  is placed in SPM
4:     if  $i \leq (F + L)$  then ▷ For code  $\mathcal{M}_i$ 
5:       Perform jump correction
6:     else ▷ For data  $\mathcal{M}_i$ 
7:       Statically allocate global data object to  $DSPM$ 
8: Perform Address Assignment
9: if  $(\mathcal{T} - \eta) \neq 0$  then
10:   Repair solution  $x$ 
11:   Repeat Steps 2–8
12: Insert memcpy() calls and call literal pool placement algorithm
13: Again perform Address Assignment to accommodate memcpy() calls and literal pool changes
14: Generate a static version of the code and perform WCET and energy analyses
15: Collect Analyses results and discard the static version of the code

```

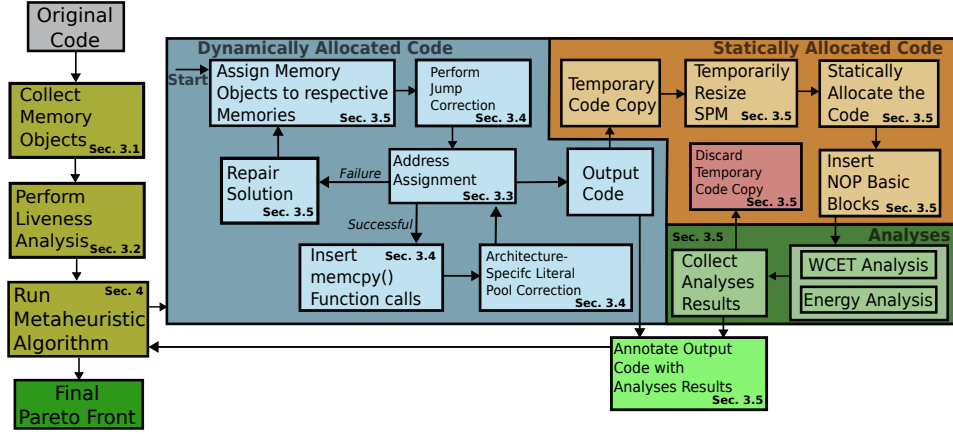
3.4 Code Transformation for Dynamic SPM Allocation

We perform code transformations during compile time to dynamically move *addrObj* from one memory to another. To copy a *addrObj* during runtime, we insert a memory copy function (*memcpy()*) at the assembly level. The *memcpy()* is allocated to the Flash, and it takes the start address (α), size (β), and destination address (δ) of an *addrObj* as inputs. The values for α , β , and δ are available after solving the address assignment problem. These three parameters enable *memcpy()* to copy code from one memory to another during runtime. After solving the address assignment problem and if $(\mathcal{T} - \eta) = 0$, we place *memcpy()* calls at the assembly level before the *memObj_c*. This call to *memcpy()* enables the program to copy code from α to δ during runtime. Once the code is copied from one memory to another during runtime, we also want our code to jump to δ instead of α . Therefore, we perform jump correction, such that previously valid jumps to α are replaced by new jumps to δ , enabling the code to jump to a proper destination address during runtime.

The code transformation for DSA is architecture-specific, i.e., we implemented these mechanisms for an ARMv7-based architecture within WCC. Placing *memcpy()* calls and jump correction code at the assembly level could increase the distance between BBs and literal pools referred to by the BBs. Therefore, we implemented an ARMv7-specific algorithm to fix literal pool placement within WCC. To fix literal pool placement, we move the literal pool near the BB referring them and generate a jump over the moved literal pool [3]. For the sake of brevity, we are skipping WCC-related implementation details and the detailed explanation of the architecture-specific code transformation for DSA. On the other hand, no major code transformation is needed for the static allocation of global data variables *memObj_d*. We assign *memObj_d* to respective memories according to their allocations at the assembly level, and an additional startup code to move *memObj_d* statically is needed.

3.5 Analyses of Dynamically Allocated Code

Performing WCET and energy analyses for dynamically allocated code at compile-time using static analysis tools is not feasible. To circumvent this problem, we generate a temporary static version of the code by virtually placing *memObj* within different memories according to the solution x . We assume that all the required *memObj* will fit within SPM, i.e., the SPM is temporarily resized to generate the static version of the code. Then, we assign *memObj*



■ **Figure 1** Dynamic SPM Allocation-based Multi-Objective Optimization Framework.

to respective memories, insert the *memcpy()* function calls at appropriate places according to the results from the address assignment algorithm, and perform jump correction. The *memcpy()* function is annotated with parametric loop bounds, which helps WCET and energy analyzers to calculate the contribution of *memcpy()* for each *memObj_c*. Therefore, WCET and energy contributions of dynamically copying *memObj_c* are collected at the compiler level.

Assigning *memObj_c* to SPM can affect the memory addresses of the remaining code in Flash. In order to keep the memory layout in Flash unchanged for static analyses, we insert *NOP* BBs in Flash in place of *memObj_c* that are statically allocated to SPM. These *NOP* BBs do not contribute to the final WCET and energy analysis results. Once the static version of the code is analyzed, we collect the results and discard this temporary code version. Algorithm (1) describes the process needed for DSA code generation and analyses. The address assignment algorithm may fail to assign all the *memObj_c* to appropriate addresses within SPM due to liveness conflicts. So, instead of discarding the whole solution, we repair the solution and then generate the dynamically allocated code for the repaired solution. We repair the solution by moving the *memObj_c* that are not assigned to an SPM address by the address assignment algorithm back to Flash.

4 Multi-Objective Dynamic SPM Allocation-based Optimization

In this section, we formulate a compiler-level DSA-based multi-objective optimization that minimizes the program’s WCET and energy consumption. Fig. (1) depicts the proposed DSA-based multi-objective optimization framework. The figure presents the flow between different aspects of DSA code generation and its analyses explained in Sec. (3). A multi-objective optimization problem performing DSA can be mathematically formulated as a minimization problem as follows.

$$\min_x F(x) = (F_1(x), F_2(x)), \quad \text{subject to} \quad x_{(F+1):(F+L)} = x_{(F+1):(F+L)} + \tau \quad (3)$$

$$(\mathcal{T} - \eta) = 0$$

where the objective function $F(x) \in \mathbb{R}^2$ represents WCET and energy consumption corresponding to a solution vector x . $x \in X$ represents a d -dimensional binary decision variable vector, where Eq. (1) describes each coordinate x_i , $i = \overline{1, d}$ and $X \subset \{0, 1\}^d$ is the search space of the DSA problem. The first constraint is applied based on the liveness analysis, i.e.,

if a function is allocated to SPM and the loops contained within that function are allocated to Flash, then it is logical that the loops contained within that function are also allocated within the SPM. Within this constraint, τ is a L -dimensional binary vector, and each element τ_l , $l = \overline{1, L}$ is '1' if l^{th} loop is in Flash and there exists a function f placed in SPM that has *liveness* conflict with the considered loop, i.e.,

$$\tau_l = \begin{cases} 1, & \text{if } x_{F+l} = 0 \ \& \ (\exists f \mid \lambda_{F+l} \subseteq \lambda_f \in \Lambda_{1:F}) \ \& \ x_f = 1 \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

where, $\lambda_{(F+l)}$ represents the set of BBs *live* for the $(F+l)^{\text{th}}$ *memObj* or l^{th} loop *memObj*, and $\Lambda_{1:F}$ is the set of sets of *live* BBs for function *memObj* (c.f. Sec. (3.2)). Furthermore, the second constraint $(\mathcal{T} - \eta) = 0$ says that the address assignment algorithm should return 0 for the solution vector x . We utilize the metaheuristic algorithms FPA and SPEA To solve the multi-objective optimization. In order to identify the trade-offs between different solutions of multi-objective optimization, we introduce a few definitions.

► **Definition 1.** Let $x_1, x_2 \in X$ and $F(x) = (F_1(x), F_2(x))$, then x_1 dominates x_2 , i.e., $x_1 \prec x_2$, if $\forall t \in \{1, 2\} F_t(x_1) \leq F_t(x_2)$ and $\exists r \in \{1, 2\} : F_r(x_1) < F_r(x_2)$.

► **Definition 2.** The solutions that are not dominated by any other solution are called Pareto optimal solutions. The set of all such Pareto optimal solutions is called Pareto optimal set, and the set of corresponding objective vectors is called the Pareto optimal front.

The initial population for FPA and SPEA are defined randomly and can influence the final Pareto front. Therefore, we perform evaluations using five different initial populations reducing the influence of the initial population on the final results. As the true Pareto front is unknown for our problem, we combine approximated Pareto fronts found by several runs of the algorithm for different initial populations into a set of nondominated points as reference Pareto front \mathcal{P} . To evaluate and compare the quality of the proposed MO_D , we use the following quality indicators:

► **Definition 3.** Coverage ($\mathcal{C} \in [0, 1]$) [26] describes the total number of dominated points in a solution set A , i.e., $\mathcal{C} = 1 - \frac{|\{a \in A : \exists p \in \mathcal{P}, a \preceq p\}|}{|A|}$

► **Definition 4.** Non-Dominated Ratio (NDR $\in [0, 1]$) [9] measures the ratio of non-dominated solutions that are contributed by a particular solution set A to the non-dominated solutions provided by all solutions sets, i.e., $NDR = \frac{|\mathcal{P} \cap A|}{|\mathcal{P}|}$

► **Definition 5.** Non-Dominated Solutions (NDS $\in [0, 1]$) [4] calculates number of non-dominated solutions concerning A itself compared to \mathcal{P} , i.e., $NDS = \frac{|\{a \in A : a \in \mathcal{P}\}|}{|A|}$

The two metaheuristic algorithms considered in this paper use three operators to explore the search space. FPA uses local and global pollination operators [25], and SPEA uses recombination and mutation operators to update each individual at every iteration [26]. They use a selection operator to collect the top-scoring solutions using the definition of Pareto dominance (c.f. Def. (1)) and pass them to the next iteration. After pre-defined stopping criteria, the algorithms output the final Pareto optimal front.

Algorithm (2) presents the DSA-based multi-objective optimization performed at the compiler level. To initialize the algorithm, we recognize and collect all the *memObj* by performing standard control flow and depth-first analyses within WCC. Then, we perform the liveness analysis to determine the *live* ranges of the said *memObj*. To perform the multi-objective optimization, we need to initialize the metaheuristic algorithm. As mentioned

Algorithm 2 Multi-Objective DSA-based optimization.

```

1: Collect memObj, perform Liveness Analysis, and randomly initialize initial population of size  $N$ 
2: for  $n = 1 : N$  do
3:   Call Algorithm (1)
4: while Stopping criteria is not reached do
5:   Update Individual using respective update operators
6:   for Each updated Individual do
7:     Call Algorithm (1)
8:   Update to next generation using selection operator
9: return Pareto-optimal solution set

```

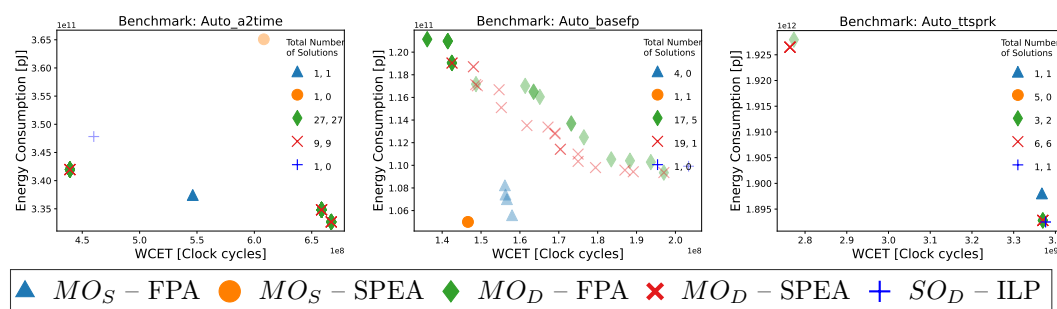
before, in this paper, we use both FPA and SPEA algorithms separately to solve the optimization problem. We randomly initialize the initial population of size N and then call the Algorithm (1) to generate dynamically allocated code and analyze the individuals. Moreover, a maximum number of generations is set as the stopping criterion, as we want the metaheuristic algorithm to terminate at some point. The metaheuristic algorithm updates the individuals at every generation and calls the Algorithm (1) to collect their objective values. Based on the objective values, the selection operator uses Pareto-dominance to select the population for the next generation. Once the stopping criterion is reached, the algorithm provides the Pareto-optimal solutions.

5 Evaluations

To the best of our knowledge, this is the first attempt to solve a compiler-level DSA-based multi-objective optimization problem that simultaneously trades multiple objectives. Therefore, we use SO_D referenced from [24] and MO_S [12] as the base for comparisons in the following evaluations. These approaches are implemented within the WCC framework, where WCET and energy analyses are performed using `aiT v21.04i` [1] and `EnergyAnalyser v21.04i` [23], respectively. WCC generates the DSA code for ARMv7-based architecture that consists of separate $ISPM$ and $DSPM$. The evaluations compare MO_D , MO_S , and SO_D in terms of the final Pareto optimal sets and the quality of obtained solutions. As SO_D is a single-objective optimization problem, we solve SO_D to minimize WCET and perform the energy analysis of the final solution to obtain both WCET and energy values.

We used benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium (EEMBC)` [18] during evaluations. While performing evaluations, we use `-O0` optimization flag to turn off any other compiler-level optimization and avoid their influence on our results. We adjust the $ISPM$ and $DSPM$ sizes individually to 60% relative to the benchmark size to increase the pressure on the optimization. Moreover, we assume that the benchmarks will fit within the Flash memory.

While solving MO_D and MO_S using FPA, the switch probability is set to 0.8, as the probability of global pollination is lower than local pollination in nature. The positive integer λ for the standard gamma function and the scaling factor γ are set to 1.5 and 0.1, respectively [25, 21]. For SPEA, the size of the external population set is set to 10, which is equal to the size of each generation's population. The recombination and mutation probabilities are set to 0.8 and 0.2, respectively. The size of the population for each generation and the maximum number of generations for both algorithms are set to 10 and 80, respectively. The results obtained during these evaluations are valid for the algorithm parameters described above. For SO_D , there is no need to set any parameters.



■ **Figure 2** Solutions Obtained from MO_S , MO_D , and SO_D optimization runs.

5.1 Pareto Fronts

Fig. (2) presents the solutions found by MO_S , MO_D , and SO_D . For the sake of brevity, this subsection presents the solutions for 3 randomly chosen benchmarks.¹ The final Pareto fronts in these figures are represented using a 2D scatter plot. The x -axis and the y -axis represent WCET and energy consumption, respectively. The legend at the bottom of the figure represents the optimization runs to which the solutions belong. The legend at the top-right corner of each subfigure shows the total number of solutions returned by respective optimization runs. The same legend indicates the number of solutions on \mathcal{P} out of the total solutions. The darker-colored solutions in the figure represent the solutions on \mathcal{P} , and the fainter version of those colors represents the solutions returned by each optimization run.

For `Auto_a2time`, all solutions obtained by MO_D -FPA and MO_D -SPEA lie on \mathcal{P} . MO_S -FPA found 1 solution that is on \mathcal{P} , and the one obtained using MO_S -SPEA is not on \mathcal{P} . The 1 solution obtained by SO_D -ILP does not lie on \mathcal{P} . As SO_D is a single-objective optimization, it always outputs a single solution in the end. In the case of `Auto_basefp`, MO_D found 17 and 19 solutions using FPA and SPEA, out of which only 5 and 1 lie on \mathcal{P} , respectively. Furthermore, MO_S -SPEA found 1 solution on \mathcal{P} , and solutions obtained MO_S -FPA and SO_D -ILP do not lie on \mathcal{P} . For `Auto_ttsprk`, except for MO_S -SPEA, other 4 optimization runs found solutions on \mathcal{P} . For this benchmark, SO_D -ILP performed better than MO_S -SPEA.

For all benchmarks, we compared the total number of solutions found by respective approaches and the total number of those solutions on their final Pareto front \mathcal{P} . In that case, MO_S using FPA and SPEA had, on average, 22.92% and 32% solutions on their \mathcal{P} , and MO_D using FPA and SPEA had, on average, 51.44% and 36.75% solutions on their \mathcal{P} , respectively. Moreover, SO_D found solutions on \mathcal{P} for only 2 benchmarks. Furthermore, we calculate the contribution of each approach to the total number of solutions on \mathcal{P} for all benchmarks. In that case, MO_S -FPA and MO_S -SPEA contributed, on average, 3.62% and 5.26% solutions, MO_D -FPA and MO_D -SPEA contributed, on average, 70.4% and 20.1% solutions, and SO_D -ILP contributed, on average 0.66% solutions to the total number of solutions on \mathcal{P} for all benchmarks.

5.2 Quality Indicators

To evaluate and compare the quality of the proposed MO_D , we use three quality indicators, namely *Coverage* (C) (c.f. Def. (3)), *Non-Dominated Ratio* (NDR) (c.f. Def. (4)), and *Non-Dominated Solutions* (NDS) (c.f. Def. (5)). From these definitions, we can say the

¹ All the remaining figures can be made available at the readers' request.

following: The lower the value of \mathcal{C} , the better the approach, and the higher the values of NDR and NDS , the better the approach. Table (1) shows \mathcal{C} , NDR , and NDS indicators for all the evaluated benchmarks. For each benchmark, the table presents the values of the quality indicators for both MO_S , MO_D , and SO_D . MO_S and MO_D used FPA and SPEA algorithms, and SO_D used ILPs to solve the optimization problem. Under each quality indicator column, we have compared their values, and for each benchmark, the better quality metric is highlighted in bold in the table.

MO_S -FPA, MO_S -SPEA, and SO_D -ILP found better or indifferent solutions in terms of \mathcal{C} for 9, 7, and 2 benchmarks, respectively. MO_D -FPA and MO_D -SPEA found better or indifferent solutions for 9 and 10 benchmarks, respectively, in terms of \mathcal{C} . Therefore, we can say that, in terms of \mathcal{C} , MO_S and MO_D using FPA performed equally, MO_D -SPEA performed the best, and SO_D -ILP performed the worst. MO_S -FPA and MO_S -SPEA found either better or indifferent solutions in terms of NDR and NDS for 5 and 7, and 9 and 7 benchmarks, respectively. MO_D -FPA and MO_D -SPEA found either better or indifferent solutions in terms of NDR and NDS for 9 and 6, and 9 and 10 benchmarks, respectively. SO_D -ILP found either better or indifferent solutions for 0 and 2 benchmarks in terms of NDR and NDS , respectively. In terms of NDR , MO_D -FPA performed best, and MO_D -SPEA performed best in terms of NDS . From overall evaluations, we can say that, in general, MO_D performed slightly better than MO_S and much better than SO_D .

Finally, we also calculated the WCET and energy overheads that the MO_D solutions incur due to the dynamic allocation of memory objects during runtime using `memcpy()`. For all benchmarks, `memcpy()` functions contributed 24.39% and 22.65% to the total WCET and energy consumption, respectively. Therefore, even with a very simple and unsophisticated implementation of the `memcpy()` function that is actively executed by the processor, we obtained slightly better quality solutions using MO_D over MO_S . Furthermore, we can see that MO_D clearly outperforms SO_D . Our next steps will focus on offloading the processor from the dynamic copying of memory objects by exploiting Direct Memory Access (DMA). This way, we expect that our proposed MO_D will outright outperform MO_S .

6 Conclusion

In this paper, we proposed a novel compiler-level DSA-based multi-objective optimization that simultaneously minimizes the WCET and energy consumption of the program. The DSA model proposed in this paper handles the dynamic movement of memory objects. Moreover, we extended the WCET and energy analyses framework within the compiler to handle analyses of such dynamically allocated code. Finally, we proposed a DSA-based multi-objective optimization framework. To solve the DSA-based multi-objective optimization problem, we used two metaheuristic algorithms, namely, FPA and SPEA. We evaluated and compared the results of the proposed optimization with MO_S and SO_D using real-world benchmark suites from EEMBC. Evaluations showed MO_D provided more solutions on the final Pareto front than MO_S . Moreover, MO_D clearly outperformed SO_D . The MO_D solutions consist of `memcpy()` overheads, still, the evaluation showed that the proposed approach can provide slightly better solutions than the well-established MO_S approach.

This paper is the first step toward compiler-level DSA-based multi-objective optimization. The next step would be to improve the approach proposed in this paper by reducing the overheads. In this paper, we saw that the overheads in terms of WCET and energy due to `memcpy()` are significant. Therefore, in the future, we will explore methods to decrease

■ **Table 1** Performance Metrics for MO_S , MO_D and SO_D .

Benchmarks	Coverage						NDR						NDS					
	MO_S		MO_D		SO_D	MO_S		MO_D		SO_D	MO_S		MO_D		SO_D			
	FPA	SPEA	FPA	SPEA	ILP	FPA	SPEA	FPA	SPEA	ILP	FPA	SPEA	FPA	SPEA	ILP			
Auto_a2time	0	1	0	0	1	0.03	0	0.73	0.24	0	1	0	1	1	0			
Auto_aifft	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_aifrf	1	1	0	1	1	0	0	1	0	0	0	0	1	0	0			
Auto_aifft	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_basefp	1	0	0.71	0.95	1	0	0.14	0.71	0.14	0	0	1	0.29	0.05	0			
Auto_bitmnp	0	1	1	1	1	1	0	0	0	0	1	0	0	0	0			
Auto_cacheb	1	0	1	1	1	1	0	1	0	0	0	1	0	0	0			
Auto_canldr	0	1	1	1	1	1	0	0	0	0	1	0	0	0	0			
Auto_idctrn	1	1	0	0	1	0	0	0.8	0.2	0	0	0	1	1	0			
Auto_iirflt	0	0.5	1	0.5	0	0.27	0.27	0	0.36	0.1	1	0.5	0	0.5	1			
Auto_matrix	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_pntrch	1	1	0	0	1	0	0	0.91	0.1	0	0	0	1	1	0			
Auto_puwmod	1	1	0	0	1	0	0	0.91	0.1	0	0	0	1	1	0			
Auto_rspeed	0	1	0	0	1	0.14	0	0.14	0.71	0	1	0	1	1	0			
Auto_tblock	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Auto_ttsprk	0	1	0.33	0	0	0.1	0	0.2	0.6	0.1	1	0	0.67	1	1			
Netw_ip_pktcheck	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Netw_ospfv2	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Netw_routelookup	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Tele_autocor	0	0	1	1	1	0.5	0.5	0	0	0	1	1	0	0	0			
Tele_conven	1	1	0	0	1	0	0	0.78	0.22	0	0	0	1	1	0			
Tele_fbital	0.5	0.33	0.28	0.5	1	0.04	0.08	0.54	0.33	0	0.5	0.67	0.72	0.5	0			
Tele_fft	0	1	0	1	1	0.5	0	0.5	0	0	1	0	1	0	0			
Tele_viterb	0	0.5	1	1	1	0.5	0.5	0	0	0	1	0.5	0	0	0			


the overhead incurred due to *memcpy()* and improve the MO_D solution quality even more. Currently, we are integrating DMA support within WCC to use it in conjunction with our DSA model for dynamically copying code during runtime.

References

- 1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2021.
- 2 Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.
- 3 ARM Limited. *ARM Compiler armasm User Guide Version 5.06*, 2010-2016.
- 4 Sanghamitra Bandyopadhyay and Arpan Mukherjee. An algorithm for many-objective optimization with reduced objective computations: A study in differential evolution. *IEEE Transactions on Evolutionary Computation*, 19(3):400–413, 2014.
- 5 Jean-Francois Deverge and Isabelle Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *ECRTS*, pages 179–190, 2007.
- 6 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.
- 7 Poletti Francesco, Paul Marchal, David Atienza, et al. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.
- 8 Michael R Garey, Ronald L Graham, and Jeffrey D Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 143–150, 1972.
- 9 Chi-Keong Goh and Kay Chen Tan. A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 13(1):103–127, 2008.
- 10 Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2008.

- 11 Shashank Jadhav and Heiko Falk. Multi-objective optimization for the compiler of real-time systems based on flower pollination algorithm. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, pages 45–48, 2019.
- 12 Shashank Jadhav and Heiko Falk. Approximating wcet and energy consumption for fast multi-objective memory allocation. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 162–172, 2022.
- 13 Mahmut Kandemir, J Ramanujam, Mary Jane Irwin, et al. Dynamic management of scratchpad memory space. In *DAC*, pages 690–695, 2001.
- 14 Yooseong Kim, David Broman, and Aviral Shrivastava. WCET-Aware Function-Level Dynamic Code Management on Scratchpad Memory. *ACM TECS*, 16(4):1–26, 2017.
- 15 Yu Liu and Wei Zhang. Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. *KIISE JCSE*, 9(2):51–72, 2015.
- 16 Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size. *Software: Practice and Experience*, 41(21):1437–1458, 2011. DOI 10.1002/spe.1079.
- 17 Kateryna Muts and Heiko Falk. Multi-Criteria Function Inlining for Hard Real-Time Systems. In *RTNS*, pages 56–66, 2020. DOI 10.1145/3394810.3394819.
- 18 Jason A Poovey, Thomas M Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE micro*, 29(5):18–29, 2009.
- 19 Robert Pyka, Christoph Faßbach, Manish Verma, et al. Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications. In *SCOPES*, 2007.
- 20 Meikang Qiu, Zhi Chen, Zhong Ming, et al. Energy-Aware Data Allocation With Hybrid Memory for Mobile Cloud Systems. *IEEE ISJ*, 11(2), 2017.
- 21 Douglas Rodrigues, Xin-She Yang, André Nunes De Souza, and João Paulo Papa. Binary flower pollination algorithm and its application to feature selection. In *Recent advances in swarm intelligence and evolutionary computation*, pages 85–100. Springer, 2015.
- 22 Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In *ECRTS*, 2017.
- 23 TeamPlay Consortium. Deliverable D4.5 - Report on Energy Usage Analysis and on Prototype - Version 1.0, 2020.
- 24 Manish Verma and Peter Marwedel. Scratchpad Overlay Approaches for Main/Scratchpad Memory Hierarchy. In *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, pages 83–119. Springer, 2007.
- 25 Xin-She Yang, Mehmet Karamanoglu, and Kingshi He. Flower pollination algorithm: a novel approach for multiobjective optimization. *Engineering Optimization*, 46(9):1222–1237, 2014.
- 26 Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*, volume 63. Citeseer, 1999.

Constant-Loop Dominators for Single-Path Code Optimization

Emad Jacob Maroun ✉ 

Institute of Computer Engineering, TU Wien, Vienna, Austria

Martin Schoeberl ✉ 

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Peter Puschner ✉ 

Institute of Computer Engineering, TU Wien, Vienna, Austria

Abstract

Single-path code is a code generation technique specifically designed for real-time systems. It guarantees that programs execute the same instruction sequence regardless of runtime conditions. Single-path code uses loop bounds to ensure all loops iterate a fixed number of times equal to their upper loop bound. When the lower and upper bounds are equal, the loop must iterate the same number of times, which we call a constant loop.

In this paper, we present the constant-loop dominance relation on control-flow graphs. It is a variation of the traditional dominance relation that considers constant loops to find basic blocks that are always executed the same number of times. Using this relation, we present an optimization that reduces the code needed to manage single-path code. Our evaluation shows significant performance improvements, with one example of up to 90%, with mostly minor effects on code size.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Theory of computation → Graph algorithms analysis; Theory of computation → Control primitives; General and reference → Performance

Keywords and phrases single-path, dominators, algorithms, optimization, control-flow graph

Digital Object Identifier 10.4230/OASICS.WCET.2023.7

Supplementary Material *Software (Source Code)*: <https://github.com/t-crest/patmos-llvm-project/tree/82eb73bff7336674027afecb254f1e3ebd1c23c2>

archived at `swh:1:rev:82eb73bff7336674027afecb254f1e3ebd1c23c2`

Acknowledgements This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

1 Introduction

Real-time systems are unique in their timing requirements. In addition to producing the correct logical results, real-time programs must produce these results within a specific time frame called the *deadline*. A result produced after the deadline is unacceptable, regardless of its logical correctness. Real-time systems must statically guarantee that a task terminates within the deadline. Here, a program's *worst-case execution time* (WCET) is the critical metric. In the simplest case, if the WCET can always be shown to be shorter than the deadline, we know that the program will always produce its result in time for it to be useful. For multi-task and multi-processor systems, scheduling must be done using each task's WCET to ensure all tasks adhere to their deadlines.

It is almost impossible to know the actual WCET of a program. Therefore, WCET analysis provides an upper bound for it. This *WCET bound* can be used instead of the real WCET when designing the system and verifying its timings. However, the halting problem



© Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner;
licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 7; pp. 7:1–7:13

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

has shown that creating a program (in this case, a WCET analyzer) that can tell whether any given program will terminate is impossible [18]. To get around this inconvenience, real-time programs are developed with certain restrictions that allow the code to be analyzable without running afoul of the halting problem. One such restriction is to have loops with a bound on the maximum number of iterations. In a real-time program, all loops must have an upper bound on the number of iterations they may perform at runtime. This is a guarantee that the programmer provides – often in the form of an annotation in the code – to the WCET analyzer, which allows the analyzer to calculate an upper bound to the execution time. A *best-case execution time* (BCET) is also often of interest for task scheduling [23]. To enable efficient BCET analysis, a lower bound on loops is also often provided such that the analyzer does not have to use zero as the default lower bound. If a loop’s lower and upper bounds are equal, we call it a *constant loop*; a loop that always executes the same number of iterations.

Single-path code generation is a code-generation technique that ensures that programs execute the same sequence of instructions regardless of runtime conditions [22]. This type of code makes WCET analysis much easier, as the analyzer does not have to account for the program executing different code traces based on what happens at runtime. The properties of single-path code can significantly affect execution time [21]. Therefore, it must be optimized to reduce the execution-time overhead.

The control-flow graph (CFG) is a directed graph that shows how execution can flow through a function.¹ Each node represents a block of sequential code, with edges specifying where execution can continue. We use *block* and *node* interchangeably in this paper. If a node in the CFG has multiple outgoing edges, we call that a branch. Depending on some runtime condition, execution continues at the target node of one edge. Loops in a function are represented by cycles in the CFG. The dominance relation identifies whether a node is guaranteed to be executed before another node. This relation is critical in compiler construction to ensure correct code generation and optimization [2]. However, the relation does not account for loop bounds and constant loops.

In this paper, we present a new CFG relation called *constant-loop dominance*. It is a variation of dominance that accounts for whether loops are constant to find blocks executed a fixed number of times. Functions called from such blocks can be optimized to reduce the overhead of converting them to single-path code. The contributions of this paper are: (1) a definition of the constant-loop dominance relation and an algorithm for calculating it; (2) a description of an optimization to single-path code that makes use of the relation; and (3) an implementation of the algorithm and optimization in a compiler that produces single-path code.

The paper is organized into six sections: The following section presents related work. Section 3 provides background information to support the understanding of the rest of the paper. Section 4 introduces the constant-loop dominance relation, an algorithm for finding constant-loop dominators, how it is used to optimize single-path code, and a brief description of the implementation. Section 5 evaluates our optimization’s performance and code size impacts. Section 6 concludes the paper.

2 Related Work

The dominance relation is fundamental within compiler construction. Its first description was given with a simple, $\mathcal{O}(n^4)$ algorithm [17]. It was used to implement global common expression elimination and loop identification. Its use continued in other important advances

¹ We do not consider inter-procedural CFGs in this paper.

like enabling the efficient computation of static single assignment form [6], which opens up further optimization opportunities [10, 24, 30]. Significant work has been put into reducing the runtime complexity of computing the dominance relation [1, 12, 29]. The state-of-the-art includes an algorithm that runs in $\mathcal{O}(m\alpha(m, n))$, where n is the number of nodes, m is the number of edges, and α is the inverse Ackermann's function [16]. Finally, the quest for a linear time algorithm has resulted in several proposals [3, 5, 9]. The challenge has been translating the theoretical runtime complexity into practical implementations that outperform the older, non-linear algorithms.

Knowledge about loop bounds is a fundamental requirement for analyzing WCET. As such, any annotation language must include the ability to specify bounds [14]. However, since manual annotations can be tedious for programmers to provide and be a source of imprecision and errors, significant effort has gone into automatic methods for finding loop bounds [4, 11, 28]. Effort has been put into finding scenarios that can automatically derive loop bounds. E.g., upper loop bounds can be derived by assuming a loop terminates and then enumerating the state-space of the variables that influence the loop exit [7]. Machine learning has also been used to try and find loop bounds [13]. While our work only uses annotated loop bounds to find constant loops, any method for finding loop bounds is compatible.

Single-path code was introduced as a code generation technique specifically for real-time systems [22]. It can be automatically generated from any WCET analyzable source code, with a significant but manageable performance cost [21]. Single-path code is challenged by its execution-time overhead. One avenue for improving this is to take advantage of its inherent instruction-level parallelism when scheduling on a VLIW architecture [20]. In [19], we extend single-path code with techniques to compensate for execution-time variability from memory accesses. This ensures that single-path code has a constant execution time, eliminating the need for WCET analysis.

3 Background

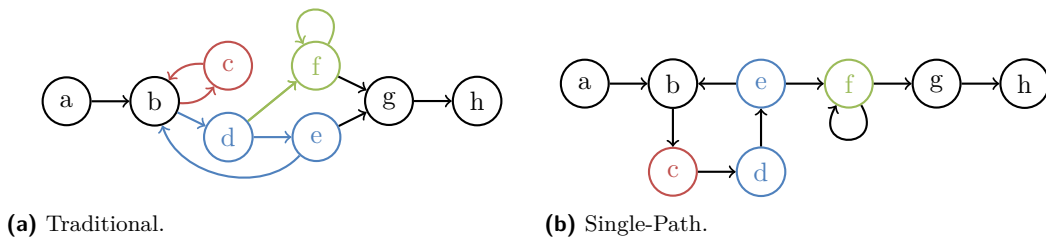
3.1 The Patmos Processor

Patmos was specifically designed for real-time systems. It is a RISC-style instruction-set architecture with features that make it time-predictable and optimized for a low WCET [27]. Patmos has an in-order, dual-issue pipeline that maximizes throughput while being time-predictable. All instructions are predicated by one of eight boolean predicate registers. If the value of the predicate is true, an instruction is *enabled*, which means it executes normally. If the predicate is false, the instruction is *disabled*. It still gets executed in the same amount of time. However, it does not read from or write to any memories or update any registers; effectively, the instruction becomes a no-op.

3.2 Single-Path Code

Single-path code was initially intended to make it computationally easier to perform WCET analysis. Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one execution path. To convert a function into single-path code, three techniques are used:

If-Conversion. Any conditional branching is converted into predicated instructions, such that only the needed path's instructions are enabled at runtime. The resulting code always executes all instructions in both paths, with only one path being enabled at a time. Looking



■ **Figure 1** Conversion of a function with branching control flow (left) to single-path code (right).

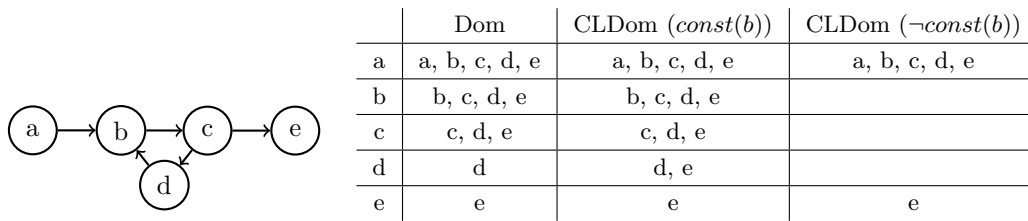
at Figure 1, we can see the result of transforming a function to single-path code. Block **b** conditionally branches to either **c** or **d**. The color coding of Figure 1a's blocks matches the conditions that led to that path being taken. In Figure 1b, the colors indicate that only if the corresponding condition is true will the block's instructions be enabled at runtime. As such, we can see how if-conversion results in **b** always leading to first **c** and then **d**. However, only if the red condition holds at runtime will **c**'s instructions be enabled. The same holds for **d**, leading to either **e** or **f** in the traditional code, but eventually leading to both in the single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.

Loop-Conversion. Loops may iterate a variable number of times depending on runtime conditions. To avoid this variability, single-path code converts loops to always iterate the maximum possible number of times. Any superfluous iterations are instead disabled using predication. Looking at our example, we can see the function has two loops, one containing the blocks **b**, **c**, **d**, and **e**, and the second containing only **f**. A single-path loop maintains a count of how many iterations have been executed and keeps looping until the maximum is reached. Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have their instructions disabled.

Function-Conversion. Single-path code also has to account for function calls. Say we have two branching paths, one of which performs a function call while the other does not. If we predicate the function call as we do for the rest of the instructions, it will not cause control to shift to the called function. This means the function's instructions are not executed (neither enabled nor disabled) if the path it was called from was disabled. Function-conversion ensures every function call is performed the same number of times, analogously to loop-conversion. Any call not logically necessary is instead disabled. Function-conversion copies all functions that are called within single-path code. The copies are then modified to take an extra predicate register argument, which specifies whether the function was called from an enabled or disabled path. The body of the copied function is then predicated on that register; if it is called from a disabled function, it will be disabled, too, and vice versa. The call instruction in the caller is not predicated, instead being provided with the predicate of the calling code to pass on. This ensures all functions are always called and executed, regardless of whether their callers are enabled or disabled.

3.3 Definitions

A *loop* in a CFG is a set of strongly connected nodes, i.e., a path exists between any two nodes. A *natural loop* additionally has an entry node, the *header*, which dominates all other nodes in the loop, and a *back edge* that enters the header from another node in the loop.



■ **Figure 2** Example CFG with the traditional dominator and constant-loop dominator relations.

The source node of a back edge is called a *latch*. An *exit edge* is an edge that connects a node in the loop to one outside of it. If two natural loops have the same header, we treat them as the same loop. We refer to these “merged” natural loops as a single *loop*. We define the number of iterations a loop performs as the number of times a path enters its header. All loops are either disjoint or nested within one another and identified by their headers. As such, every node has a header, which is the header of the innermost loop it is contained in. For consistency, we also consider the entire function as a pseudo-loop with the entry node as the header. Nodes without successors are *end nodes* and are assumed to return from the function.

Removing all back edges from the CFG results in an acyclic graph called a *forward control-flow graph* (FCFG). We partition the FCFG into *loop FCFGs* of the subgraphs containing only nodes whose header is the loop header. This means that for each loop, we now have a dedicated FCFG. Each node in the graph is only in one FCFG, except the headers, which reside in the FCFG of their enclosing loop and in the FCFG of the loop for which they are headers. Exit edges are represented as edges from the header of the inner loop to the original target node in the outer loop.

4 Constant-Loop Dominance

We consider CFGs with an optional label, *const*, on the headers of loops. If the label is present, it means paths through the loop header must visit the header a fixed number of times before exiting its loop. Otherwise, the number of visits may fluctuate between different paths. We define the constant-loop dominance relation as follows: **A node x constant-loop dominates a node y ($x \text{ cldom } y$) if every path from the entry to y visits x a fixed number of non-zero times.**

Looking at Figure 2, we can see the traditional dominator relation (Dom) and the constant-loop dominator relations (CLDom) for the given CFG. CLDom is shown with the loop headed by *b* being both constant and variable. The most obvious difference is that when the loop is variable, none of its nodes constant-loop dominate other nodes or themselves. If the loop is constant, we can see the result is the same except *d* also constant-loop dominates *e*, unlike with traditional dominance. The last iteration of the loop must exit through *c*, meaning *d* is always visited $i - 1$ times, where i is the max iteration count. Note that constant-loop dominators behave the same as traditional dominators in the absence of loops.

4.1 Algorithm

Finding traditional dominators on *acyclic directed graphs* (DAGs) is done by calculating the dominance for each node in topological order. Using topological order ensures that the dominance of a node’s predecessors is established before getting their intersection to result in the dominance of the current node (and remembering to add self-dominance.)

Algorithm 1 Constant-Loop Dominators.

```

CLDom(s):                                     ▷ Starting node as input
1:  $H \leftarrow$  Inner loop headers
2:  $D \leftarrow \emptyset$                        ▷ Dominator set for nodes of  $fcfg(s)$ 
3:  $ID \leftarrow \emptyset$                     ▷ Dominator sets for inner loops
4:  $IED \leftarrow \emptyset$                  ▷ End-Dominator sets for inner loops
5: for  $h$  in  $H$  do                         ▷ Analyze inner loops
6:    $ID[h], IED[h] \leftarrow CLDom(h)$ 
7: end for
8: for  $b \mid b \in topological\_sort(fcfg(s))$  do   ▷ Find dominators
9:    $P \leftarrow \bigcap \{D[a] \cup IED[a] \mid \forall (a, b) \in fcfg(s) \wedge a \in H \wedge const(a)\} \cap$ 
      $\bigcap \{D[a] \mid \forall (a, b) \in fcfg(s) \wedge (a \notin H \vee \neg const(a))\}$ 
10:   $D[b] \leftarrow \begin{cases} P \cup b & \text{if } b \notin H \vee const(b) \\ P & \text{otherwise} \end{cases}$ 
11: end for
12: for  $h, v \mid \forall h \in H \wedge \forall v \in ID[h]$  do   ▷ Extract dominators from loops
13:   $D[v] \leftarrow \begin{cases} D[h] \cup ID[h][v] & \text{if } const(h) \\ D[h] & \text{otherwise} \end{cases}$ 
14: end for
15:  $L \leftarrow \bigcap \{header\_end\_dominators(l, D, IED) \mid \forall (l, s)\}$ 
16:  $E \leftarrow \bigcap \{header\_end\_dominators(e, D, IED) \mid \forall (e, c) \in exit\_edges(s)\}$ 
17:  $C \leftarrow \{c \mid \forall c \notin exits(s) \wedge \forall e \in exits(s) \wedge c \in header\_end\_dominators(e, D, IED)\}$ 
18: return  $D, (L \cap (E \cup C))$ 

```

This traditional algorithm is the basis for our algorithm for finding constant-loop dominators. It can be seen in Algorithm 1 on lines 8-11, where P (the intersection of predecessors) has been edited for our relation. Instead of operating on the CFG, our algorithm operates on the FCFG of the start node ($fcfg(s)$), which is a DAG. Since traditional and constant-loop dominance are equivalent when there are no loops, they are also equivalent between pairs of nodes within the same FCFG. This baseline, therefore, finds the correct constant-loop dominance between nodes in the same FCFG. The rest of the algorithm accounts for dominance between nodes of different loops (nested or in sequence).

In addition to returning the constant-loop dominator sets for each node in the given FCFG, CLDom returns a second, helper set we will call the *end dominators*. The set is calculated on lines 15-18. It represents the set of nodes in the current FCFG that would constant-loop dominate a hypothetical successor node to the FCFG's loop – assuming that node did not have any other predecessors. It is calculated by finding the nodes that constant-loop dominate all latches (L) and constant-loop dominate all exits (E) or are strictly constant-loop dominated by all exits (C). The nodes adhering to these requirements are precisely those that will always be visited a fixed number of times; they either are visited in every iteration ($L \cap E$) or will be skipped in the last iteration only ($L \cap C$). The helper function *header_end_dominators* does the following: If the given node is in the FCFG ($n \in fcfg(s)$), the function returns that node's constant-loop dominators ($D[n]$). Otherwise, the node must be in one of the inner loops. *header_end_dominators* finds the header in the FCFG ($h \in H$) whose loop contains the node; either directly or in a nested loop. If that header is constant, it returns the constant-loop dominators of the header and end dominators of that inner loop ($D[h] \cup IED[h]$). Otherwise, it returns the header's constant-loop dominators alone ($D[h]$).

Our algorithm starts by recursing on the headers of inner loops in the FCFG (lines 5-7) and storing the results for each. During dominator calculation for each FCFG node, we add the end dominators of any constant headers to their dominator sets before intersecting with the other predecessors ($D[a] \cup IED[a]$). This is what ensures that any constant-loop dominators are extracted from inner loops into the dominator sets of the current loop's nodes. Notice that *header_end_dominators* serves the same purpose in the end-dominators calculation.

Lastly, we also need to extract the constant-loop dominators of the nodes of inner loops into the current dominator set (lines 12-14). We give the loops' nodes the dominators of the header in the current dominator set, as those would not have been available in the recursive call (since *fcfg(h)* does not contain nodes from outside the loop.) For constant loops, we also add the dominator sets of each node from their loop's recursive call ($ID[h][v]$) so they are included in the final result. Not doing so for variable loops ensures that nodes within a variable loop do not dominate anything, not even themselves.

To use CLDom for getting the constant-loop dominators of a function, we call it on the entry node and ignore the end-dominators result. It will always be empty since functions have no latches or exits.

4.2 Pseudo-Root Optimization

Single-path code can take advantage of constant-loop dominators to reduce execution times. The optimization focuses on those blocks that constant-loop dominate all end blocks, which means they will always be executed the same number of times per function call. We will refer to these blocks as *constant-loop dominant*.

As described in Section 3, function-conversion makes functions in single-path code take a predicate argument to enable or disable their bodies. However, this is not always necessary. This is most obvious for any single-path *root function*; a function that is itself single-path but is called from a non-single-path function (e.g., the main function.) A root function is guaranteed to be enabled, making the predicate argument unnecessary. The original single-path implementation recognized this and special-cased root functions not to need the predicate argument [21]. This reduces the number of instructions needed for predicate management, which results in reduced execution time.

The optimization of root functions can also be used for other functions. Any function that we can guarantee is always called enabled can be optimized as if it was a root. Taking this further, any function called from a constant-loop dominant block can also be optimized. We can do so because it means we know exactly how many times the function is called from that point, and function-conversion therefore does not need to account for variations in call numbers (as there is no variation). The callee in cases like these is called a *pseudo-root* since its code generation can be identical to a root's. Any function called from a root or pseudo-root in a constant-loop dominant block is also a pseudo-root.

The pseudo-root optimization uses constant-loop dominance to explore the call tree from the root function(s) and identifies all pseudo-root functions. The single-path transformation then uses the information to optimize all pseudo-roots to omit the predicate argument. It also changes all call instructions to pseudo-roots to be predicated, so the functions are not called when a block is disabled. Note that a function may be called from both a constant-loop dominant block and one that does not dominate. E.g., it could be called both in the entry block of a function and within only one side of a branch. In such cases, functions are duplicated, such that two versions are used: one that takes an additional argument and one that does not.

4.3 Implementation

We extend the open-source work presented in [21] with implementations of the constant-loop dominance algorithm and the described optimization. Patmos’ compiler is based on the LLVM compiler framework [15]. Its frontend, called Clang, produces the LLVM intermediate representation called Bitcode. Bitcode is then compiled by the backend into machine code. The previous work and our extensions all reside in the backend.

We have implemented our algorithm as a `MachineFunctionPass` in the LLVM backend. At this stage, functions are in an intermediate representation close to Patmos machine code. Our algorithm is run on each function and returns a map from their blocks to the set of blocks that constant-loop dominate them. Our CFG does not have a `const` label. Instead, we provide the algorithm a function that, when given a header, returns whether it should be treated as constant. It does so by looking at the loop iteration bounds; if they are equal, the loop must be constant.

Identifying pseudo-roots is done in the `SPMark` pass of the single-path transformation [21]. We update it so that while identifying functions that will be called from a single-path context, it also identifies which calls are coming from a constant-loop dominant block and marks the target functions as pseudo-roots.

The `SPReduce` pass assigns each instruction its predicate. It also removes predication from call instructions and provides the additional predicate argument to functions. When it sees a call instruction in a constant-loop dominant block in a pseudo-root function, it omits the predicate argument, predicates the call instruction, and targets the pseudo-root version of the function (instead of the version that takes a predicate argument.)

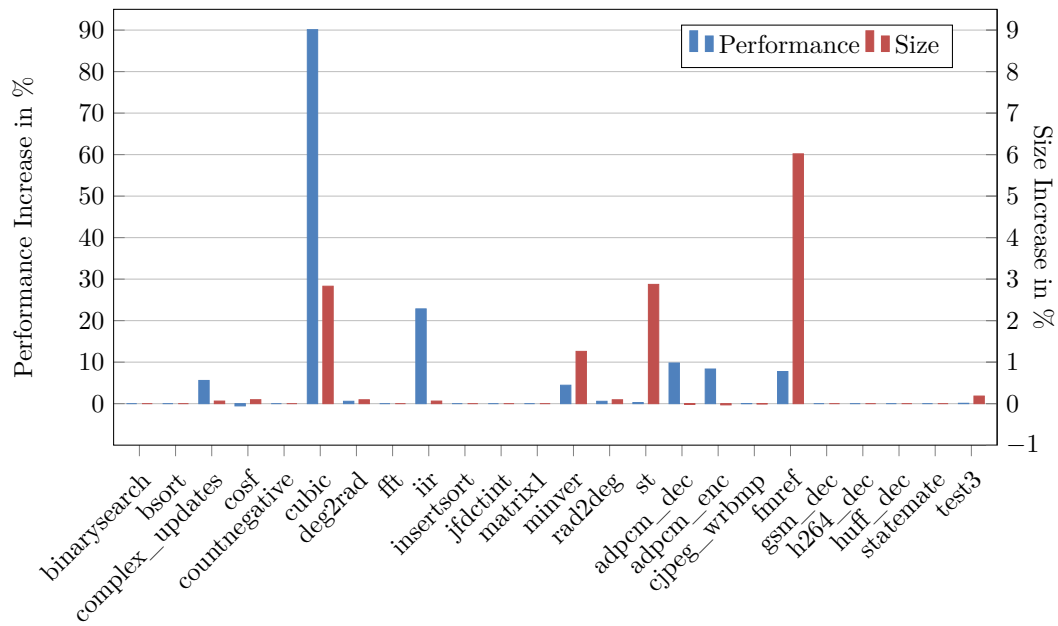
5 Evaluation

We use a subset of the TACLe benchmark suite [8] to evaluate the effect of enabling the pseudo-root optimization for single-path code. We only include those programs that compile and run correctly for single-path code with and without the optimization. We exclude the `duff` program, as it has no branching and is fully inlined by the compiler, meaning no changes are made to it by the single-path transformation. The `filterbank` program is also excluded because it is so long-running that the simulator we use to run all the programs, Pasim, saturates its cycle counter, meaning we do not know what the execution times are.

5.1 Performance

In Figure 3, we show the performance increase (blue bars) of enabling the pseudo-root optimization ($\frac{\text{disabled} - \text{enabled}}{\text{enabled}} \times 100$). First, note how 11 programs see no execution time differences. All these programs – except `huff_dec` and `gsm_dec` – only have one function. This can be seen in the first row of Table 1, where nine programs only have one function with and without our optimization. The second row shows how many functions were recognized as pseudo-roots. For all these functions, including `huff_dec` and `gsm_dec`, only the root was recognized as a pseudo-root, meaning there is nothing to optimize.

Enabling the pseudo-root optimization produces wildly different results for the other programs. In the lower end, `cosf` sees a small performance decrease. Looking at the third row of Table 1, we see that many more instructions are used by single-path code with the optimization (600 \rightarrow 726). The fourth row also shows an increase in the total number of call instructions (159 \rightarrow 181), while the fifth row shows that there are very few calls between pseudo-roots (8). This must mean the five pseudo-root functions found did not make up



■ **Figure 3** Performance and code size increase of enabling the pseudo-root optimization for single-path code.

for the increase in code size from duplicating three of them. On the other hand, we have the `cubic` program, which sees a 90 % performance increase. This number is all the more impressive when we look at Table 1. First, notice that the number of functions increases from 33 to 47. Notice also that the number of pseudo-roots found was 17 (including the root). This means that 14 pseudo-roots are also used in a non-pseudo-root context, which means two copies of each original function must be used. The rest of the functions are either only used in a pseudo-root context (3) or in a non-pseudo-root context (16). The additional copies of some functions also translate to an increased total of instructions used for managing the single-path code ($627 \rightarrow 921$) and the number of total call instructions ($136 \rightarrow 215$), with 58 calls being between pseudo-roots. So from where does all that performance come? The source code shows that the main function is four constant loops nested within each other. The function `cubic_solveCubic` is called four times before the loop and once in each iteration of the inner-most nested loop. Cumulatively, the main function has 879 calls to this function, all from constant-loop dominant blocks. Therefore, recognizing `cubic_solveCubic` exclusively as a pseudo-root likely produces most of this substantial increase in performance.

5.2 Code Size

As we have explained earlier and seen in our results so far, using the pseudo-root optimization can increase code size. The first source of this increase is the additional copies of functions used in both pseudo-root contexts and non-pseudo-root contexts. Code size can also be reduced when functions are exclusively pseudo-roots and therefore need fewer instructions for managing predicates and calling other pseudo-roots.

We measure the total size of the final executable of each program with and without the optimization and can see the result in the red bars of Figure 3. We can see that the difference is negligible for most of the programs that were affected by our optimization. For others,

■ **Table 1** Compiler statistics for each program using single-path code. For each entry, the pseudo-root optimization is disabled for the upper number and enabled for the lower. The metrics given are the total number of functions, the number of pseudo-root (PR) functions, the number of single-path management instructions, the total number of call instructions, and the number of calls between pseudo-roots.

	binary..	bsort	complex..	cosf	countn..	cubic	deg2rad	fft	iir	insert..	jfdctint	matrix1	minver	rad2deg	st	ad..dec	ad..enc	cjpeg..	fmref	gsm..	h264..	huff..	state..	test3
Functions	1	1	21	30	1	33	27	1	21	1	1	1	33	27	34	3	3	3	72	3	1	2	1	101
PRs	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Instructions	19	32	302	600	32	627	433	91	303	34	20	40	827	433	546	112	142	71	1458	389	116	199	47	1510
Calls	0	0	64	159	0	136	87	0	65	0	0	0	116	87	121	4	4	4	377	8	0	5	0	200
PR-Calls	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

there is a significant increase but none prohibitively so. We can see no correlation between the increase in performance and code size. E.g., while `cubic` sees an enormous performance increase, it only sees a 2.8 % size increase. `fmref`, on the other hand, sees a 6 % size increase for a comparatively modest 7.7 % performance increase.

Lastly, we also need to note that the executables we have measured do not exclusively contain single-path code. They also include all original versions of any single-path function, any initialization code that eventually calls the benchmark function, and the standard library. This means the size differences are likely bigger for both increases and decreases in a real-world, single-path-only scenario.

5.3 Source Access

Patmos and its platform, T-CREST [25], are available as open-source and include the contributions of this paper. The Patmos homepage can be found at <http://patmos.compute.dtu.dk/> and provides a link to the Patmos Reference Handbook [26], which includes build instructions.

The T-CREST project repositories can be found at <https://github.com/t-crest>, with the repository for the compiler used in this work at <https://github.com/t-crest/patmos-llvm-project> (commit hash: 82eb73bff7336674027afecb254f1e3ebd1c23c2).

6 Conclusion

In this paper, we presented the constant-loop dominance relation and how it can be used for optimizing single-path code. We first defined the relation as a variation of the traditional dominance where the number of visits to a node must be constant. This takes loop bounds into account to recognize constant loops. We then presented a recursive algorithm for finding the constant-loop dominators. It first explores (nested) loops and uses the intermediate results for the outer loops. We showed how the relation can be used to identify pseudo-root functions in single-path code. These have the quality of being called a fixed number of times. We used this property to optimize single-path code to require fewer instructions to manage predicates and to reduce unnecessary calls. Our evaluation showed sporadic but significant performance improvements from applying our optimization. While some programs saw no execution-time differences, others saw an up to 90 % performance increase. We also showed that the optimizations do affect code size, with executable sizes increasing by up to 6 %.

References

- 1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 253–265. ACM, 1973. doi:10.1145/800125.804056.
- 2 Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1977.
- 3 Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM J. Comput.*, 28(6):2117–2132, 1999. doi:10.1137/S0097539797317263.
- 4 Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. oRange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, volume 42, 2008.
- 5 Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008. doi:10.1137/070693217.
- 6 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi:10.1145/115372.115320.
- 7 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 Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 6 of *OASICs*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL: <http://drops.dagstuhl.de/opus/volltexte/2007/1194>.
- 8 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICs*, pages 2:1–2:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASICs.WCET.2016.2.
- 9 Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 185–194. ACM, 1985. doi:10.1145/22145.22166.
- 10 Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 97–105. ACM, 1998. doi:10.1145/277650.277668.
- 11 Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real Time Syst.*, 18(2/3):129–156, 2000. doi:10.1023/A:1008189014032.
- 12 Paul Walton Purdom Jr. and Edward F. Moore. Immediate predominators in a directed graph [H] (algorithm 430). *Commun. ACM*, 15(8):777–778, 1972. doi:10.1145/361532.361566.
- 13 Dimitar Kazakov and Iain Bate. Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning. In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2006, September 20-22, 2006, Diplomat Hotel Prague, Czech Republic*, pages 421–428. IEEE, 2006. doi:10.1109/ETFA.2006.355425.

- 14 Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Softw. Syst. Model.*, 10(3):411–437, 2011. doi:10.1007/s10270-010-0161-0.
- 15 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.
- 16 Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. doi:10.1145/357062.357071.
- 17 Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969. doi:10.1145/362835.362838.
- 18 Salvador Lucas. The origins of the halting problem. *J. Log. Algebraic Methods Program.*, 121:100687, 2021. doi:10.1016/j.jlamp.2021.100687.
- 19 Emad J. Maroun, Martin Schoeberl, and Peter Puschner. Compiler-directed constant execution time on flat memory systems. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023.
- 20 Emad J. Maroun, Martin Schoeberl, and Peter P. Puschner. Compiling for time-predictability with dual-issue single-path code. *J. Syst. Archit.*, 118:102230, 2021. doi:10.1016/j.sysarc.2021.102230.
- 21 Daniel Prokesch, Stefan Hepp, and Peter P. Puschner. A generator for time-predictable code. In *IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC 2015, Auckland, New Zealand, 13-17 April, 2015*, pages 27–34. IEEE Computer Society, 2015. doi:10.1109/ISORC.2015.40.
- 22 Peter P. Puschner and Alan Burns. Writing temporally predictable code. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), 7-9 January 2002, San Diego, CA, USA*, pages 85–94. IEEE Computer Society, 2002. doi:10.1109/WORDS.2002.1000040.
- 23 Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings*, pages 165–172. IEEE Computer Society, 2002. doi:10.1109/EMRTS.2002.1019196.
- 24 Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27. ACM Press, 1988. doi:10.1145/73560.73562.
- 25 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil C. Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: time-predictable multi-core architecture for embedded systems. *J. Syst. Archit.*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.
- 26 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.
- 27 Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real Time Syst.*, 54(2):389–423, 2018. doi:10.1007/s11241-018-9300-4.
- 28 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.

- 29 Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974. doi:10.1137/0203006.
- 30 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. doi:10.1145/103135.103136.

Analyzing the Stability of Relative Performance Differences Between Cloud and Embedded Environments

Rumen Rumenov Kolev ✉

TTTech Auto AG, Wien, Austria

TU Wien, Austria

Christopher Helpa ✉

TTTech Auto AG, Wien, Austria

Abstract

There has been a shift towards the software-defined vehicle in the automotive industry in recent years. In order to enable the correct behaviour of critical as well as non-critical software functions, like those found in Autonomous Driving/Driver Assistance subsystems, extensive software testing needs to be performed. The usage of embedded hardware for these tests is either very expensive or takes a prohibitively long time in relation to the fast development cycles in the industry. To reduce development bottlenecks, test frameworks executed in cloud environments that leverage the scalability of the cloud are an essential part of the development process. However, relying on more performant cloud hardware for the majority of tests means that performance problems will only become apparent in later development phases when software is deployed to the real target. However, if the performance relation between executing in the cloud and on the embedded target can be approximated with sufficient precision, the expressiveness of the executed tests can be improved. Moreover, as a fully integrated system consists of a large number of software components that, at any given time, exhibit an unknown mix of best-/average-/worst-case behaviour, it is critical to know whether the performance relation differs depending on the inputs. In this paper, we examine the relative performance differences between a physical ARM-based chipset and a cloud-based ARM-based virtual machine, using a generic benchmark and 2 algorithms representative of typical automotive workloads, modified to generate best-/average-/worst-case behaviour in a reproducible and controlled way and assess the performance differences. We determine that the performance difference factor is between 1.8 and 3.6 for synthetic benchmarks and around 2.0-2.8 for more representative benchmarks. These results indicate that it may be possible to relate cloud to embedded performance with acceptable precision, especially when workload characterization is taken into account.

2012 ACM Subject Classification Software and its engineering → Software development techniques

Keywords and phrases Performance Benchmarking, Performance Factor Stability, Software Development, Cloud Computing, WCET

Digital Object Identifier 10.4230/OASICS.WCET.2023.8

1 Introduction

In recent years, the task complexity of automotive software has increased, and the quantity and quality of software has grown proportionally. To put this software complexity into perspective, a premium car today has more than 100 million object code instructions [6]. The development of this volume of embedded software runs into its own unique challenges. One such challenge is the need for embedded hardware to be available to the developers. Since the requirements for the software are formulated according to the embedded chipset, most performance and functional benchmarks and tests must be executed on the specific embedded hardware to determine the behaviour of safety-critical real-time systems. Even for non-safety-critical real-time applications, it is necessary to assess the overall load/latency of



© Rumen Rumenov Kolev and Christopher Helpa;
licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wagemann; Article No. 8; pp. 8:1–8:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

integrated components. Modern automotive software systems are distributed, with many software components contributing to the execution time of an algorithm at the same time. Thus, the overall WCET depends on the performance of multiple different components, yet it is not necessarily composed of the sum of the WCETs of the separate components. Instead, execution times are context-dependent, as discussed in [21]. Benchmarks, such as the ones introduced in this paper, use this context dependency to their advantage by utilizing real-life use cases to measure and compare performance.

Due to hardware and licensing costs, as well as legal and logistical problems, uninterrupted access to target hardware is rarely available to every developer. This is a critical issue since the inability to perform quick verification and tests of the code leads to a decrease in efficiency and quality. Furthermore, safety requirements require extensive testing of autonomous driving applications. It is not feasible to execute all those tests on dedicated physical hardware, and thus, cloud technology could provide a sensible solution to this issue due to its vertical (scale up/down where changes occur in the resources of a VM/container) and horizontal (scale in/out that adds/removes VMs/containers) scalability [2]. This is sufficient for functional tests but not for assessing correctness regarding temporal behaviour. The temporal behaviour can lead to different functional behaviour, leading to tests not representative of real behaviour [11].

Autonomous driving applications have been tested in the cloud, using simulations, yet the behaviour has been shown to differ from the track testing, in part due to timing issues [8]. Hence, testing entirely in the cloud has not been fully providing the required test coverage. Recently, ARM hardware became available in the cloud [14], bringing it closer to embedded hardware, which is often ARM-based as well, due to ARM's energy efficiency and high performance. This allows more similar timing. However, differences in the chip implementation, like the micro-architecture and memory subsystem layout, will lead to performance differences depending on the nature of the workloads running on the machines [19], [12].

The goal of this paper is to determine first how the performance of the cloud and the embedded hardware can be related and, secondly, whether a simple factor would be expressive enough to represent this relation.

To treat performance in the cloud as a reliable indicator of performance on the embedded hardware, using a performance difference factor ($= Perf_{embedded}/Perf_{VM}$), two conditions need to be met. First, the factor jitter must be relatively low for repeating test executions and must not be overly affected by timing outliers. Second, the factor must not differ significantly based on different inputs to the algorithms (e.g., a factor is valid for all cases from the best- to the worst-case behaviour). Thus, we investigate the stability of relative performance differences of benchmarks over the entire input range and for different workloads, executed on a cloud VM (running on dedicated hardware) and on embedded hardware and perform a root cause analysis of any observed outliers. The environments differ not only in the chip used but also in the software infrastructure, e.g., the hypervisor present in the cloud and potential differences in the operating systems. We expect that certain timing outliers exist, which would need to be taken into account in any future framework that might try to relate the cloud to embedded performance.

Should a stable factor be identified, two questions can be answered early on in the development process by analyzing the cloud performance exclusively: Would the set of all applications that should be hosted together fit into the embedded CPU, or can it already be foreseen that they are too compute-intensive? How realistic is it that the real-time applications would meet their end-to-end timing requirements on the embedded hardware?

The rest of the paper is structured as follows: Section 2 provides the relevant background

information and related work, Section 3 discusses the three benchmarks and the most important implementation details and input variation of the two representative benchmarks, Section 4 presents an overview of the benchmarking environment, as well as of the different benchmark executions and an interpretation of the results, Section 5 provides a final overview of the results and a summary of the open points.

2 Background and Related work

Modern general-purpose microprocessors on the market differ in features like clock frequency, cache sizes and structure, core count, memory bandwidth, power consumption, and, in general, their architecture and micro-architecture. However, these specifications are not enough to reliably compare performance differences, as CPU performance is highly dependent on the specific workloads being executed, e.g., a higher clock frequency does not imply a better performance of the memory subsystem. Additionally, discrepancies in the performance factor might be caused by differences in the branch prediction algorithms, timing anomalies in the instruction pipeline, etc.

While considerable research has been done in recent years in relation to predicting new workloads or CPU/GPU performance (c.f. [20], [1]), it has focused on the average case. Thus, this paper aims to extend the state of the art by analyzing the performance difference factors of the best- and worst-case performance. One particular problem is how to reliably stimulate representative best-, average- and worst-case behaviour of the applications.

Most applications hosted on automotive ECUs are best-case, soft real-time (infotainment), or firm real-time systems (autonomous driving), and the deployed software is typically a combination of both. No static WCET analysis is performed due to the overly pessimistic results, especially on modern multi-core SoCs and the extensive efforts required [18]. The uncertainty of measurement results over static WCET approaches can be tolerated, as the non-hard real-time context does not require strict guarantees [21]. A low number of deadline misses can typically be tolerated in autonomous driving functions (e.g., cruise control). In this sense, autonomous driving applications are an example of firm real-time systems. Measurement-based analysis has much lower analysis efforts and leads to less hardware overprovisioning need, which would lead to prohibitive hardware costs. Therefore, this type of analysis is useful for the industry use case, where it can speed up development and reduce costs. However, some low-level causes for timing anomalies, which could lead to unexpected and hard-to-predict execution time outliers that could make it infeasible to predict the performance difference, are known and discussed by Lundqvist and Stenstrom in [12], e.g., out of order instruction execution causing domino effects. These types of anomalies were not identified in the performed experiments and are therefore not discussed in further detail. Other, higher-level anomalies are possible, e.g., preemptions due to real-time throttling caused by a FIFO scheduler. This is touched upon in Section 4.

3 Benchmark Details

General overview. While selecting benchmarks for the analysis, we focused on software relevant to the automotive field. Besides that, the main requirement is the ability to control the best-/average-/worst-case performance by varying the input.

Two algorithms were selected - a pathfinding algorithm (the A* search algorithm) and a contour detection algorithm. The A* algorithm, or variations of it, is used for path planning in mapping software. Contour detection algorithms are a vital part of the object extraction

8:4 Stability of Relative Performance Differences

process, e.g., used for object recognition in autonomous driving.

A third, more generic benchmark was selected - EEMBC CoreMark-PRO. It utilizes a combination of integer and floating-point workloads and large datasets to stress the entire processor [7]. Due to these features, it provides a range of performance difference factors for workloads with isolated processor stress points and hence gives upper/lower bounds for factors to expect. The two representative algorithmic benchmarks exhibit a more realistic combination of different stress points.

A* algorithm. The A* algorithm was implemented according to its definition [9].

Our implementation of the algorithm works using Manhattan-style grids, represented as 2D C++ vectors, where each “cell” of the grid corresponds to a vector element. Grids are generated in the benchmark using a C++ standard library random number generator in order to block some cells to reduce possible paths and make the search environment more realistic. Maximum grid size and the seed of the RNG function are configured using input parameters. Grid generation is not timed; exclusively, the algorithm execution is.

The heuristic functions used to modify the performance of the algorithm are depicted in Table 1. Worth noting is that inconsistent, yet admissible, heuristics have been shown to expand arbitrarily more cells than an alternative A*-like algorithm in [4].

■ **Table 1** Heuristic functions used in the A* benchmark and their effects on its performance.

heuristic function	best-case	Euclidean distance	constant	inconsistent
algorithm behaviour	uses precalculated values for the actual distance to the goal cell	calculates the Euclidean distance to the goal cell	always estimates distance 0 to the goal	randomly estimates the distance between 0 and Euclidean distance to the goal (thus, admissible)
performance impact	best case	average case, good estimate, thus close to best-case	average case, bad estimate	worst-case

Contour detection. The contour detection benchmark was implemented following the OpenCV documentation [16]. In the timed section, the input image is read, converted to grayscale, binary thresholding is applied, contours are found, and contours are drawn.

Three classes of 5120x2880px input images were selected to trigger the best-/average-/worst-case performance of the algorithm, depicted in Table 2. An additional fourth set of worst-case type images was added, with 8k resolution, to analyze the behaviour when increasing image resolution. Fifteen images of each class are used. The contour approximation mode and the contour retrieval mode also cause differences in performance [17], [15].

4 Benchmark Executions and Results

4.1 General

Environment. The code for both representative benchmarks is written in C++17 and compiled using g++ 9.4.0 for Ubuntu. For the contour detection benchmark, OpenCV 4.6.0 is used. It is important to note that the shared object files for the C++ STD and OpenCV and the binaries for each benchmark are shared between the two platforms to avoid any performance differences caused by different compiler versions/settings.

■ **Table 2** Inputs for the contour detection benchmark and their effects on its performance.

image classes				
image type	solid-color images	dashcam pictures of street traffic	multicolored white noise	
algorithm behaviour	only image borders are considered contours	average real-life use case	very large number of neighbouring pixels with different colours => highest possible number of contours	
performance impact	best case	average case	worst case	
contour approximation method				
method	CHAIN_APPROX_SIMPLE		CHAIN_APPROX_NONE	
performance impact	removes redundant points: less memory intensive	less	all the boundary points are stored: very memory intensive	
contour retrieval mode				
mode	EXTERNAL	LIST	CCOMP	TREE
performance impact	least compute-/memory-intensive	no hierarchy: less memory-intensive	2-level hierarchy: slightly more memory-intensive	all hierarchy levels: most memory-/compute-intensive

For the cloud virtual machine, a standard Microsoft Azure type D2pds v5 was chosen, which provides an entire physical core for each vCPU. The embedded hardware consists of a Janicto TDA4VMx&DRA829Vx processor connected to a J721EXCP01EVM common processor board. The specifications [3], [10] of both are listed in Table 3.

These platforms closely represent a real-world development scenario in the automotive industry. The embedded CPU is “ARMv8-A”-ISA-based, while the VM utilizes a newer “ARMv8.2+”-ISA. The cloud CPU also supports a higher clock frequency and a larger cache, including a 32MB system-level cache, which is not present on the embedded CPU. Thus, it is to be expected that both the computationally intensive (due to clock frequency) and the memory-intensive (due to memory and caching model [13]) performance of the VM would be significantly better.

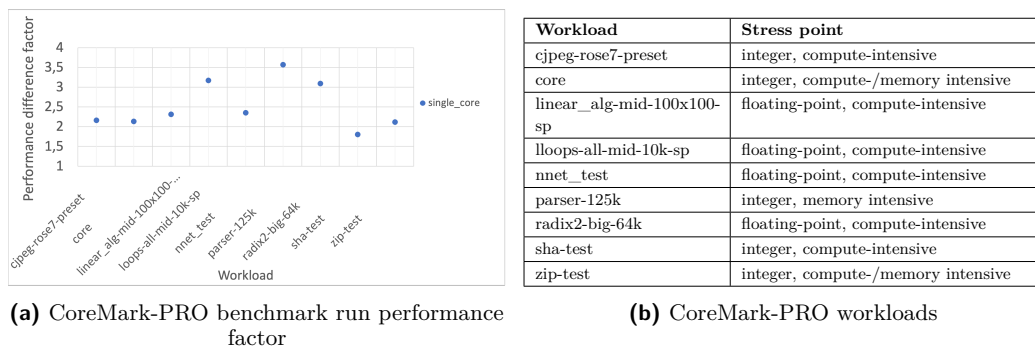
Due to the fact that the virtual machine utilizes a hypervisor and the different Linux distributions in use, we are, in reality, not analyzing the performance differences between the two different ARM chips but the whole stack (OS, hypervisor, hardware). This is acceptable and even desirable, as it is the environment that would be present in the industry-relevant use case.

Preliminary analysis. The A* benchmark was configured to execute 10 times for every heuristic and grid size, and the contour detection benchmark was configured to execute 20 times for every image and contour retrieval/approximation mode. The entire benchmarks were executed multiple times with little to no variance in the results.

A preliminary analysis was performed to evaluate how stable the execution environment is. Some potential temporal anomaly sources were identified, like scheduling and background loads. Therefore, for both the A* and contour detection benchmark executions, it was investigated whether the CPU affinity or the scheduling algorithm has any effect on the performance factor. The taskset command was used to assign the CPU affinity to a single core. The FIFO scheduling algorithm was enabled, once with maximum and once with above-average priority for the benchmark process. No significant difference in the performance factor was observed, except for some outliers caused by the FIFO scheduler throttling the OS and being forced to reschedule due to the priority of the benchmark being maximized. As these outliers do not occur with the default scheduler and the root cause is known, plots for these runs are omitted. This indicates that the measurements are highly reproducible, and there are no apparent external effects that introduce timing anomalies into the applications.

4.2 EEMBC CoreMark®-PRO

A run of CoreMark-PRO was performed to observe the behaviour of the performance difference factor when the CPU is stressed in different ways. The plot in Figure 1 depicts the results. A short description of the stress point of each workload is presented in the table in Figure 1. [7] provides more detailed descriptions of each workload. The average performance factor for



■ **Figure 1** CoreMark-PRO benchmark execution.

CoreMark-PRO workloads varies between 1.8 and 3.6, averaging around 2.52. Of interest are the 3 outliers as seen in the plot in Figure 1.

The first increase to above a factor of 3 is the workload “loops-all-mid-10k-sp”. This workload is part of the floating point suite. It is computationally intensive, as it carries out different mathematical kernels. The second outlier occurs with the integer-based workload “parser-125k”, which is created in such a way that the focus of the benchmark is data structure creation and traversal [7]. Thus, it is memory intensive. The third increase occurs within the “radix2-big-64k” workload, which is also part of the floating point suite and is computationally intensive. These results show that different performance relations exist based on the workload (integer vs floating point and compute- vs memory-intensive).

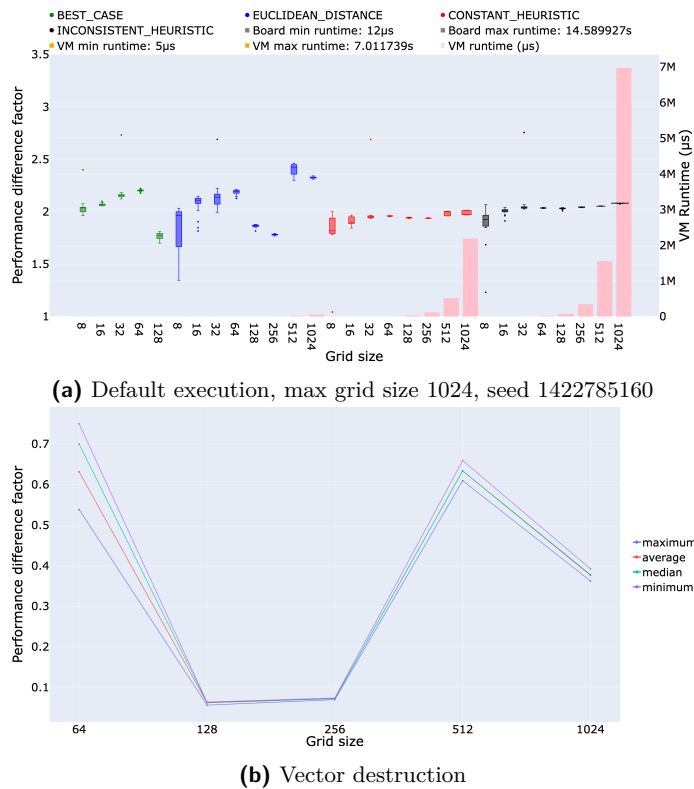
Since neither the A*, nor the contour detection benchmark is heavy on floating point computations, that is left as a possible area to explore in the future, using different benchmarks. On the other hand, both of those create, traverse and delete data structures, which leads to larger than average for the benchmarks performance difference factors for larger data structure sizes, similar to the outliers observed in the XML parser benchmark in the CoreMark suite.

4.3 A* algorithm

The A* algorithm benchmark was executed with 4 different seeds, with every heuristic. For every seed and grid size, 20 different grids were generated and tested. The same sequence of grids is generated when using the same seed, so executions using different heuristics use identical grids.

Only the plots for one of the 4 different seeds are shown, as the performance (factor) for the other 3 is analogous. Runs for grid sizes above 128 using the best-case heuristic were omitted, as these take the longest amount of time in the preparation phase, and the Euclidean distance heuristic behaves analogously due to their very similar runtimes.

Default benchmark executions. As depicted in plot (a) of Figure 2, the performance difference factor varies between 1.8 and 2.4 for the median of the runs for every combination



■ **Figure 2** A* benchmark execution and vector destruction measurements.

of heuristic, seed, and grid size. Two interesting points: First, some outliers with factor 2.7 for grid size 32 can be observed. Second, there is a decrease in the median performance factor to 1.9 and 1.8, respectively, for grid sizes 128 and 256, for Euclidean distance and best-case heuristics. Besides those outliers, the performance difference factor appears to be fairly stable.

Average performance factor outliers for grid sizes 128 and 256. The root cause of the outlier lies in the overhead of the return of the function. At the start of the function, 3 data structures are allocated. Relevant for this discussion is a 2D standard library vector of elements of type *cell*, which is a *struct* containing 2 integers and 3 doubles. This vector is of size *gridSize * gridSize* and is initialized with empty *cells*.

The cause for the decrease in performance was determined to be the destruction of this vector by using a complementary benchmark, which calls a function, which allocates a 2D *cells* vector of variable size and then returns. Measurements were performed from the point after the allocation was done to after the function had returned.

In plot (b) of Figure 2, it can be observed that the VM performs consistently worse than the embedded CPU for vector destruction, with a significant decrease in performance for grid sizes 128 and 256. We are not able to identify the root cause of this outlying behaviour. This slowdown of the VM explains the decrease in the average performance for these grid sizes for the Euclidean distance and best-case heuristics, as these are the fastest heuristics, where vector destruction takes a larger percentage of the overall execution time of the benchmark (24.8% and 30% respectively for grid sizes 128 and 256 on the VM). On the other hand, the

8:8 Stability of Relative Performance Differences

constant and inconsistent heuristics take a longer time to execute, and the vector destruction takes a smaller percentage of the overall execution time of the benchmark (1% and 0.5% respectively for constant and inconsistent heuristics, grid size 128 on VM). On the embedded hardware, the vector deletion takes less than 1.5% of the overall execution time for any heuristic. This means that there is a 23% to 29% extra execution time for these grid sizes and heuristics on the VM, which corresponds to the 23% to 29% decrease in the performance factor.

Performance factor increases for minimum case. To analyse the root cause of the outliers for grid size 32, it was determined via debugging that these occur during minimum execution times caused by a specific edge case in the algorithm. In each iteration, the A* algorithm attempts to determine the neighbouring cell with the shortest path so far and the heuristically determined shortest remaining path to the goal cell. An edge case exists, where all cells neighbouring the start cell are blocked. Therefore, the goal is immediately unreachable and no more iterations can be done, so the algorithm returns, leading to the minimal execution time. These were determined to be the cases where the performance difference factor increases.

The pattern repeats because every run represented on a single plot uses the same seed to generate grids, and the heuristic functions are never actually called in these minimum cases, as all surrounding cells are blocked.

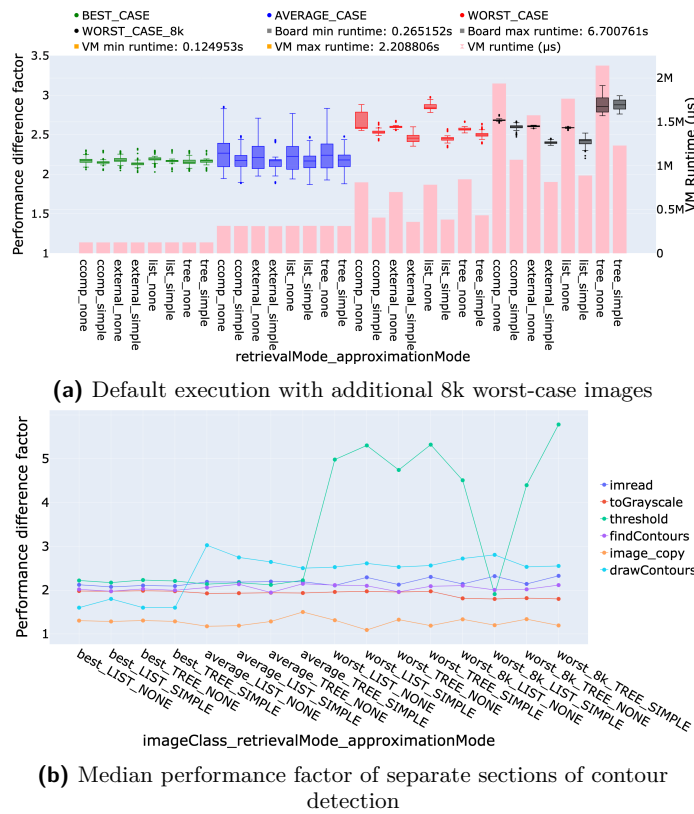
To analyze this behaviour, additional measurements of only the “surrounded” case were performed, inside and outside the function call, as well as of only the data structure initializations. Listing 1 depicts the three pairs of measurement points.

As the performance difference factor remains on average 2.2 in all of these measurements, this points towards caching differences. As the VM has a larger cache and an updated memory model [13], some differences in caching/memory operations performance are expected. This is a very probable cause for the increased factor in the minimum case of the default execution. I.e., in the default runs of the benchmark, many memory operations, which use the cache, are performed in iterations before the “surrounded” case. This leads to cache misses in the “surrounded” case. These misses do not occur in the isolated edge case measurements, as the cache does not get overwritten since every separate iteration only does the vector initialization with empty data. Thus, a higher performance factor in the “surrounded” cases during default runs is expected compared to the targeted “surrounded” case runs. The higher factor is in line with the results from other memory-intensive measurements, as discussed in e.g., Subsection 4.2 or Subsection 4.4.

4.4 Contour Detection

Default benchmark executions. Plot (a) in Figure 3 shows that the contour detection benchmark displays a median performance factor of 2.1 to 2.2 for the best- and average-case inputs. A jitter of about 0.1 is present and can be explained by the differences between the different pairs of contour retrieval and approximation methods. However, an increase up to a factor of about 2.6 can be observed for the worst-case input image class. This corresponds to a 15%-20% increase in the performance factor.

To investigate the increase in performance factor between the average- and worst-case results, two approaches were used. First, a further set of white noise images with a larger (7680×4320) resolution was introduced to analyze whether the performance factor continues to increase with increasing image resolution due to e.g., differences in caching/memory subsystem or the branch prediction algorithm. Second, the separate sections of the contour detection algorithm were timed to determine specific areas that might explain the difference.



■ **Figure 3** Contour detection benchmark executions.

8k images run. It is evident from plot (a) of Figure 3 that the average performance factor does not continue to increase with increasing image resolution. It stays in the same range of 2.4 to 2.8, as it does for the 5k worst-case images.

A single outlier run using the 8k resolution worst-case images was observed, where the average performance factor decreased to 2.2 for those images, caused by a slowdown in the performance on the VM. The slowdown was not reproducible, and therefore, the outlier is not discussed further.

Timing of separate sections of the algorithm. As can be read from plot (b) of Figure 3, the binary thresholding section of the algorithm experiences an increase in performance factor from a stable 2.2 for the best- and average-case input images to 4.5-5 for the worst-case type images. The factor increases by about 250%. The other slight increase in performance factor is seen in the image read function, which has an increase in factor of about 0.2, which corresponds to a 9%-10% increase.

The thresholding function corresponds to about 2% of the complete execution time of the algorithm, while the reading of the image accounts for about 30% of the overall runtime. Thus, the increase in performance factor caused by the reading and thresholding comes out to about 10% total increase in runtime. We attribute these increases to memory model and caching differences. It should be noted that a 5K image has a size of 14.7MB when loaded into memory. Respectively, an 8K image has a size of 33.1MB. Both of these do not fit in the L1 and L2 DCache of either CPU. However, the VM utilizes a so-called system-level cache, SLC, of size 32MB. This additional caching level likely contributes to the better performance of the VM for the worst-case image class, as cache misses that would go directly to the main

memory on the embedded CPU would, in most cases, go to the faster SLC of the cloud CPU. The worst-case image type induces more memory operations, especially in the binary thresholding section, where less optimization (e.g., through branch prediction) is possible due to the random nature of white noise. Thus, the better memory/caching model of the VM leads to better performance for these image types. The cachegrind tool [5] of the valgrind suite was used to simulate the different caching behaviour with or without the 32MB SLC of the VM by setting the last level cache as either 32MB or 1MB. The tool shows a 10% and 25% lower miss rate with the SLC, respectively, for the worst-case and worst-case-8k image classes. Cachegrind provides only a basic simulation of the caching behaviour, but the results appear to support the hypothesis that the 15% to 20% increase in the performance factor for the worst-case image classes is caused by caching differences. A profiler like perf that accesses hardware counters is a better choice for more accurate measurements, but the hypervisor of the cloud VM does not allow perf to read the counters.

5 Conclusion and Outlook

In conclusion, the A* benchmark shows an average factor of 2.045, with a standard deviation of 0.15. The contour detection benchmark shows an average factor of 2.31, with a standard deviation of 0.2. Lastly, CoreMark-PRO shows an average factor of 2.52, with a standard deviation of 0.56. The minimum and maximum factors for each of the benchmarks tend to move in the same range with some outliers of around 10%.

Overall, values between 1.8 and 3.6 were observed for the performance factor. For every workload, however, a separate range within this overall range can be observed, meaning that the factor range depends on the specific workload. The factor is consistent for every specific input type, i.e., no significant jitter occurs due to, e.g., timing anomalies. The difference between the average factors of the two representative benchmarks (A* and contour detection), which exhibit a combination of (integer only) compute- and memory-intensive operations, is around 13% with no vast outliers. These results indicate that it may be possible to use a single factor to relate the performance between the cloud VM and the embedded CPU with 10%-15% accuracy for this type of workload. Characterizing the workload regarding whether it is integer- or floating-point-intensive and computationally or memory-intensive could improve the accuracy.

Some deviation of the performance difference factor for a single algorithm is possible due to different input types causing the algorithm to transition from being compute-bound to memory-bound, thus changing the processor feature under stress. An example of this is the difference between the best-/average- and worst-case image type for the contour detection benchmark, where the worst-case type is more memory-intensive, as described in Section 4.4.

Some interesting topics for further research have also been identified in Section 4. There is always the possibility to add more benchmarks to increase the coverage of different types of workloads, stressing different aspects of the CPU. For example, a comprehensive floating point benchmark and a complex data structure benchmark would be of particular interest.

Another approach would be to investigate whether a similarly stable performance difference factor is present for GPUs or between x86-based CPUs and ARM-based embedded CPUs.

A topic of interest would be to analyze the stability of the performance of the cloud VM, as motivated by the single outlier with images of 8K resolution, mentioned in Subsection 4.4.

References

- 1 Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, 2015.

- 2 Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR)*, 51(3):1–40, 2018.
- 3 Ampere Computing. Ampere® Altra® 64-bit multi-core processor features. https://d1o0i0v5q51p8h.cloudfront.net/ampere/live/assets/documents/Altra_Rev_A1_DS_v1.30_20220728.pdf. Accessed: 2023-04-21.
- 4 Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, July 1985. doi:10.1145/3828.3830.
- 5 Valgrind Developers. Cachegrind: a high-precision tracing profiler. <https://valgrind.org/docs/manual/cg-manual.html/>. Accessed: 2023-05-07.
- 6 Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009. doi:10.1109/MC.2009.118.
- 7 EEMBC. CoreMark-PRO. <https://github.com/eembc/coremark-pro>. Accessed: 2023-03-07.
- 8 Daniel J Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Brusio, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2020.
- 9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.
- 10 Texas Instruments. DRA829 Jacinto. https://www.ti.com/lit/ds/symlink/dra829v.pdf?ts=1673940212016&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FDRA829V. Accessed: 2023-04-21.
- 11 ISO 21448:2022 Road vehicles – Safety of the intended functionality, April 2022.
- 12 Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled micro-processors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 12–21. IEEE, 1999.
- 13 Berenice Mann. Arm Architecture – Armv8.2-A evolution and delivery. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-armv8-2-a-evolution-and-delivery>. Accessed: 2023-06-18.
- 14 Paul Nash. Azure virtual machines with Ampere Altra ARM-based processors – generally available. URL: <https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available/>.
- 15 OpenCV. ContourApproximationModes. https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga4303f45752694956374734a03c54d5ff. Accessed: 2023-01-16.
- 16 OpenCV. OpenCV contour detection. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. Accessed: 2023-04-25.
- 17 OpenCV. RetrievalModes. https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga819779b9857cc2f8601e6526a3a5bc71. Accessed: 2023-01-16.
- 18 Stefan Schaefer, Bernhard Scholz, Stefan M Petters, and Gernot Heiser. Static analysis support for measurement-based WCET analysis. *Editors: Timothy Bourke and Stefan M. Petters Work-in-Progress-Chair: Liu Xiang, Peking University, China*, page 25, 2006.
- 19 Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or CPU performance by analyzing public datasets. *ACM Trans. Archit. Code Optim.*, 15(4), January 2019. doi:10.1145/3284127.
- 20 Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or CPU performance by analyzing public datasets. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–21, 2019.
- 21 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

A Tables and Listings

■ **Table 3** Benchmarking environment specifications.

	Cloud VM	Embedded CPU
ARM CPU	Ampere Altra 64-Bit Multi-Core Processor (ARM v8.2+)	64-bit Dual-core Arm Cortex-A72 (ARMv8-A)
Number of cores	2	2
max. clock frequency	3300MHz	2000MHz
L1 cache	64KB DCache, 64KB ICache per core	32KB DCache, 48KB ICache per core
L2 cache	1MB per core	1MB shared per dual-core cluster
System-Level Cache (SLC)	32MB	-
RAM	8GiB	3.8GiB(4GiB), 512KB on-chip SRAM in MAIN domain
Operating System	Ubuntu 20.04.5 LTS, Kernel: 5.15.0-1034-azure	Arago 2021.09 (based on Yocto Linux), Kernel: 5.10.65-gdcc6bedb2c

■ **Listing 1** Minimum case measurements pseudo-code

```

a_star_benchmark():
    benchmark_preparation
    time_measurement_start_OUTER
    a_star_algorithm_func():
        time_measurement_start_INNER
        time_measurement_start_INITS
        data structure inits
        time_measurement_end_INITS
        calculations
        time_measurement_end_INNER
    return result
time_measurement_end_OUTER

```




EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications

Simon Wegener  

AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany

Kris K. Nikov   

University of Bristol, UK

Jose Nunez-Yanez   

Linköping University, Sweden

Kerstin Eder   

University of Bristol, UK

Abstract

This paper presents **EnergyAnalyzer**, a code-level static analysis tool for estimating the energy consumption of embedded software based on statically predictable hardware events. The tool utilises techniques usually used for worst-case execution time (WCET) analysis together with bespoke energy models developed for two predictable architectures – the ARM Cortex-M0 and the Gaisler LEON3 – to perform energy usage analysis. **EnergyAnalyzer** has been applied in various use cases, such as selecting candidates for an optimised convolutional neural network, analysing the energy consumption of a camera pill prototype, and analysing the energy consumption of satellite communications software. The tool was developed as part of a larger project called TeamPlay, which aimed to provide a toolchain for developing embedded applications where energy properties are first-class citizens, allowing the developer to reflect directly on these properties at the source code level. The analysis capabilities of **EnergyAnalyzer** are validated across a large number of benchmarks for the two target architectures and the results show that the statically estimated energy consumption has, with a few exceptions, less than 1% difference compared to the underlying empirical energy models which have been validated on real hardware.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Software and its engineering → Automated static analysis

Keywords and phrases Energy Modelling, Static Analysis, Gaisler LEON3, ARM Cortex-M0

Digital Object Identifier 10.4230/OASICS.WCET.2023.9

Funding This work was supported by the H2020 project TeamPlay, Grant Agreement No. 779882.

Acknowledgements The authors like to thank Marcos Martinez de Alejandro, Nikos Fragoulis, Ali Sahafi, and Vangelis Vassalos for their work on the use cases and Heiko Falk and Shashank Jadhav for the integration of **EnergyAnalyzer** into WCC.

1 Introduction

Safety-critical embedded systems are used in various domains such as transportation, aerospace, medical devices, and industrial control systems. These systems are designed to meet certain non-functional requirements, such as timing or energy usage constraints, in addition to functional requirements. The satisfaction of these non-functional requirements is essential for the correct operation of the system and the safety of its users. Failure to meet these requirements can result in catastrophic consequences, such as loss of life or severe financial losses. Therefore, it is necessary to ensure that these systems are designed and implemented with reliable guarantees for their non-functional requirements.



© Simon Wegener, Kris K. Nikov, Jose Nunez-Yanez, and Kerstin Eder; licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 9; pp. 9:1–9:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For timing constraints, reliable guarantees can be obtained by using sound timing analysis methods. Timing analysis is a technique used to analyse the temporal behaviour of a system and predict the worst-case execution time (WCET) of tasks and other timing properties. Accurately determining a bound for the WCET of a task is essential for ensuring that a system meets its timing constraints and avoiding potential hazards.

Energy consumption is another crucial non-functional requirement for embedded systems. Energy usage constraints are becoming more and more important due to the increasing use of battery-powered and energy-constrained devices. Moreover, reducing energy consumption can increase the lifetime of the device and reduce its operating costs. However, ensuring that a system meets its energy usage constraints is a challenging task, as energy consumption is highly dependent on the system's workload, input data, and hardware characteristics. In contrast to timing analysis, which has a well-established theoretical foundation, creating an energy model that yields safe yet tight bounds for energy consumption is almost impossible. There are two primary reasons for this. First, energy consumption is measured in physical units (Joule), whereas processor cycles are a logical unit of time. Moreover, the amount of energy consumed by a processor is highly specific to the actual device. Two processors from the same production batch may already show a small difference in energy consumption. Additionally, the amount of energy consumed by one and the same processor may increase over time as the silicon degrades [11]. While this is true for timing as well, as the clock frequency may differ slightly from processor to processor, there is an easy mitigation: The logical unit of time (processor cycles) can be converted to the physical unit of time by multiplying with the interval of clock frequencies. Second, the actual amount of energy consumed depends on the switching activity in the processor, which is highly data-dependent. Thus, creating an energy model requires measuring all possible input combinations for each instruction, which is usually not feasible [16]. To address these limitations, several research works proposed using empirical methods to characterise energy models. For example, Georgiou et al. [9] suggest using pseudo-randomly created data to characterise an Instruction Set Architecture (ISA) energy model. This approach reduces the number of input combinations needed to create the energy model and allows for faster evaluation of the model.

In recent years, researchers have proposed using event counters to create more accurate energy models for predictable architectures. Event counters are hardware components that count the number of times certain events occur during execution, such as instructions executed or cache misses. By using event counters to create an energy model, the model can accurately capture the energy consumption of a more diverse set of programs. Furthermore, event counters are available on many modern processors, which makes the proposed energy models more accessible to developers. Pallister et al. [23] proposed an event counter-based method for data-dependent energy modelling, which is a more accurate way of modelling energy consumption for systems that process variable data sets. The proposed method identifies the relationship between the input data and the processor's energy consumption and uses this relationship to create an energy model. The authors evaluated their method on two different processors and found that it provided more accurate predictions of energy consumption than previous methods.

This paper presents **EnergyAnalyzer**, a novel tool for static energy consumption analysis. It incorporates accurate energy models [19, 20] for two specific architectures: the Gaisler LEON3 microprocessor [3], a radiation-tolerant microprocessor commonly used in the space communications sector, and the ARM Cortex-M0 microcontroller [1], known for its ultra-low power capabilities. Main contribution of this paper is the utilisation of standard techniques from WCET analysis for static worst-case energy consumption (WCEC) analysis. The

microarchitectural analysis statically predicts the progression of performance counter values which are used as input for the aforementioned energy models. When validated against model predictions using real-time samples, the static analysis shows <1% difference in estimated energy for the vast majority of tested benchmarks.

2 Related Work

Several studies have attempted to construct worst-case energy models capable of capturing the WCEC at the ISA level [13, 32]. In Jayaseelan et al. [13], the authors bound the WCEC on a simulated processor by maximizing the switching activity factor for each simulated component to obtain a WCEC cost for each ISA instruction. Although this method retrieves the WCEC for all the ISA instructions, it could result in significant overestimation because the absolute worst-case on the hardware simulation used for the energy model's characterization phase might be infeasible to be triggered by any program on the actual hardware implementation of the same architecture. Additionally, the approach is not feasible on physical hardware because there is no practical way of maximizing the switching activity on hardware. To construct an equivalent ISA energy model for a fabricated processor, one would need to exhaustively search all combinations of valid data for the operands of an instruction, making it infeasible in most cases due to the huge space of possible input data combinations for each ISA instruction. Wägemann et al. [32] constructed an energy model capable of capturing the WCEC. The specifics of the model's characterization are presented in [28]. Nevertheless, they tested this energy model on a benchmark and admitted that such an absolute energy model could lead to significant overestimations, making the retrieved energy consumption bounds less useful.

Ideally, data-sensitive energy models would be created to capture the energy cost of executing an instruction based on the circuit switching activity caused by the operands used. Such models can potentially capture the WCEC of a program without overestimation. However, recent work has demonstrated that finding the data that triggers the WCEC is an NP-hard problem and that no practical method can approximate tight energy consumption upper bounds within any level of confidence [16]. Therefore, Georgiou et al. [9] suggested using pseudo-randomly created data to characterise an ISA energy model, as their empirical evidence showed that such models tend to be close to the actual worst case. Although this approach is expected to yield loose upper-bound energy consumption estimations, their experimental results showed a low level of underestimation of the WCEC (less than 4%) for the programs tested. Such estimations can still provide valuable guidance to the application programmer to compare coding styles or algorithms in terms of resource consumption. Design decisions can be made based on empirical investigations to determine the level of over-provisioning that ensures the required level of dependability for the given application.

3 Tool Overview

Over the last several years, a more or less standard architecture for static timing-analysis tools has emerged [34, 12, 6, 30], which is also implemented in AbsInt's WCET analyser aiT. One can distinguish four major building blocks:

1. The decoder translates the executable to an internal form that is used by the other parts (value analysis, microarchitectural analysis, etc). Architecture specific patterns decide whether an instruction is a call, branch, return, or just an ordinary instruction. This knowledge is used to form the basic blocks of the control-flow graph (CFG). Then, the

- control-flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user needs to write some annotations to guide the control-flow reconstruction.
2. Afterwards, the value analysis determines safe approximations of the values of processor registers and memory cells for every program point and execution context. These approximations are used to determine bounds on the iteration numbers of loops and information about the addresses of memory accesses. Value analysis information is also used to identify conditions that are always true or always false. Such knowledge is used to infer that certain program parts are never executed and therefore do not contribute to the worst-case resource consumption. Value analysis is again architecture-dependent.
 3. The microarchitectural analysis then determines upper bounds for the execution times of basic blocks by performing an abstract interpretation of the program execution on the particular architecture, taking into account its pipeline, caches, memory buses, and attached peripheral devices. The microarchitectural analysis is even more architecture-dependent than the decoder and value analysis, as the specification of the ISA alone does not suffice to create an abstract model of the hardware's timing behaviour, but the particular specifics of a particular processor implementing this specification must be taken into account (e.g., cache size, buffers, pre-fetching, etc). The microarchitectural analysis is usually a composition of both pipeline and cache analysis.
 4. Using the results of the preceding analysis phases, the path analysis phase searches for the worst-case execution path. The analysis translates the control-flow graph with the basic block timing bounds determined by the microarchitectural analysis and the loop bounds derived by the value analysis into an Integer Linear Program (ILP). The solution of the ILP yields a worst-case path together with a safe upper bound of the WCET. Path analysis is generic, i.e., does not depend on the target architecture.

The structure of **EnergyAnalyzer** is similar to the structure of **aiT**. In fact, both tools share most components. In particular, they both use the same decoder for CFG reconstruction and the same value, loop, control-flow, and path analyses. Only the microarchitectural analysis differs. **aiT** uses a microarchitectural timing model to derive safe upper bounds of the WCET for each instruction. In contrast, **EnergyAnalyzer**'s microarchitectural analysis computes worst-case performance counter values for each basic block, which are used as input values for the microarchitectural energy models presented in Section 4. While the input values for the energy models are conservatively predicted, the model itself does not give a worst-case guarantee. During path analysis, worst-case execution frequencies of basic blocks are combined with approximate energy model results to produce a tight estimate of the worst-case energy behaviour which is not necessarily an upper bound.

4 Microarchitectural Energy Analysis

A key component of **EnergyAnalyzer** is the underlying use of accurate energy models for the target microarchitectures. Based on previous research experience, a hardware event-based methodology was utilised to generate and evaluate the models [18, 17]. Such techniques are well established and provide high prediction accuracy for both CPU and full system modelling. In their research, Rodrigues et al. [25] conducted a systematic review of Performance Monitoring Unit (PMU) events also referred to as Performance Monitoring Counters (PMCs) commonly used in modern microprocessors. They showed the effectiveness of these events in characterising and modelling dynamic power consumption. Several other studies have also explored accurate power modelling [21, 24, 33, 27, 17].

The PMC-based energy consumption estimation models were obtained via Ordinary Least Squares [14] linear regression analysis, where coefficients, β_x , are determined for each counter, C_x , to predict the overall energy cost, i.e., $E = \sum_x (\beta_x \times C_x) + \alpha$, with α being the residual error term and x being the event that is tracked by the performance counter. The coefficients β_x are the constants in the energy model that are program independent while the counters C_x are the variables that depend on the program and its input. For a specific program with known counter values, the energy model can be used to estimate the energy consumed during the program's execution.

For static-analysis-based energy consumption estimation, the overall energy consumption estimate of a piece of code is typically constructed from the estimates of the ISA basic blocks of the program. Thus, a PMC-based energy model can enable energy consumption estimation via static analysis only if the counters used for the modelling and prediction can be statically predicted at the ISA basic block level. The microarchitectural analysis of **EnergyAnalyzer** models the parts of the pipeline that have an effect on the performance counters. For example, the decode unit tracks the number of executed instructions of each type, and the load/store unit tracks the number of read and write accesses to each memory that is covered by a performance counter. Thus, for each instruction in the CFG, the microarchitectural analysis predicts an upper bound for the increase of the various performance counter values. These values are summed up for the basic blocks and then used as input for the energy model. In order to make the model scalable for block-level static analysis, we enforced a residual $\alpha = 0$. This means that at time 0 the energy predicted is also 0J. We have also used a Non-Negative Least Squares (NNLS) solver to guarantee positive weights for all the events in the final energy model, thus always guaranteeing predictable positive energy consumption values from the model at discrete time slices [15]. We apply the energy model on the worst-case performance counter values for each block.

The accuracy of the model has been evaluated by using PMC data from a test set with the generated model equations. The measured power or energy values are then compared to the estimations obtained from the model. The percentage difference or mean absolute percentage error (MAPE) between them can be used as an objective metric to quantify model accuracy. Several different models for each platform were identified and the best performing ones were integrated into **EnergyAnalyzer**. Additional details on model generation and validation techniques used for both target platforms can be found in the accompanying papers [20, 19].

4.1 ARM Cortex-M0 Setup and Energy Model

The target platform on which the Cortex-M0 models were developed and validated is the STM32F0-Discovery board, which features the STM32F051 microprocessor [29]. The platform does not feature an on-chip PMU. Thus, a special methodology was developed to obtain the necessary PMC information, using an extended version of the **Thumbulator** instruction set simulator [7]. The target platform allows ten different configurations, depending on CPU frequency, wait states for flash memory access, and whether instruction pre-fetch is enabled or not.

We selected the energy consumption model of the ARM Cortex-M0 below for integration into **EnergyAnalyzer**. It uses six statically predictable PMCs, and the resulting energy estimation is measured in nJ. The model offers an estimation error to physical measurements, calculated as Mean Absolute Percentage Error (MAPE), of 2.8%, for all the data points used for training and validation. Further details on how the energy analysis models have been generated including a breakdown of the available hardware configurations and associated model weights and performance are presented in Nikov et al. [19].

$$\begin{aligned}
E_{\text{Cortex-M0}} = & 0.972565030 \times C_{\text{executed instructions without multiplications}} \\
& + 0.652871770 \times C_{\text{RAM data reads}} \\
& + 1.031341343 \times C_{\text{RAM writes}} \\
& + 1.037625441 \times C_{\text{Flash data reads}} \\
& + 1.354953706 \times C_{\text{taken branches}} \\
& + 2.274650563 \times C_{\text{multiplication instructions}}
\end{aligned}$$

The microarchitectural analysis uses the address intervals computed by the value analysis phase to determine whether a memory read targets the RAM or the Flash memory (or possibly both). The number of load and store operations, as well as the number of taken branches, are predicted by analysing the flow of an instruction through the processor pipeline. There, the type of an instruction is also taken into account.

4.2 Gaisler LEON3 Setup and Energy Model

The LEON3 energy models were trained and validated on the GR712RC evaluation board [2]. Similarly to the STM32F0-Discovery board, this platform also does not feature a PMU. In order to get the PMC measurements for the models, a new, dual-platform approach using a Kintex UltraScale FPGA board was developed. The programmable platform was loaded with a synthesised version of the LEON3 coupled together with the LEON3 Statistics Unit (L3STAT [4]). The results were synchronised with physical sensor measurements from the GR712RC platform to obtain the complete data set for model generation and validation. More details on the platform setup, methodology, and estimation results are presented in Nikov et al. [20].

The models presented in that paper describe *fine-grained* power models which are trained and validated on all available samples. The models integrated into **EnergyAnalyzer** use the same methodology, but with one key difference: the samples in the data set are aggregated for each benchmark to create code-block-sized models, making them more *coarse-grained* and the NNLS solver is used to generate positive model weights. Since average power models would not be very helpful for this purpose, total energy consumption is used instead.

L3STAT provides several performance counters that are useful for modelling the energy consumption [5]. However, not all of them are statically predictable. Those that can be statically predicted by **EnergyAnalyzer** with high accuracy are shown in Table 1. Whether a memory access results in a possible cache miss is predicted by the cache analysis that is part of the microarchitectural analysis. The type of instructions and consequently, the update of the respective counters, are tracked in the pipeline analysis. The following energy model based on the *ISA+Cache* subset has been selected for integration into **EnergyAnalyzer** using the methodology shown in Appendix A. It has a MAPE of <8.3% compared to physical measurements and provides energy estimations in J:

$$\begin{aligned}
E_{\text{LEON3}} = & 3.93365 \times 10^{-08} \times C_{\text{integer instructions}} \\
& + 1.87111 \times 10^{-07} \times C_{\text{store instructions}}
\end{aligned}$$

■ **Table 1** *ISA+Cache* subset of PMCs.

#	Counter	Description	#	Counter	Description
C_1	ICMISS	instruction cache misses	C_{13}	TYPE2	type 2 instructions
C_3	DCMISS	data cache misses	C_{14}	LDST	load and store instructions
C_7	IINST	integer instructions	C_{15}	LOAD	load instructions
C_{11}	BRANCH	branch instructions	C_{16}	STORE	store instructions
C_{12}	CALL	call instructions			

5 Evaluation

We integrated the energy models from Section 4 into **EnergyAnalyzer for ARM Cortex-M0** and **EnergyAnalyzer for LEON3**, respectively. We evaluated the integration with the help of the BEEBS benchmark suite [22]. The goal of the evaluation is to determine how close the statically estimated energy consumption for a given workload is to the model estimation. Not all of the BEEBS benchmarks exercise the worst-case path through the program during execution. Thus, a comparison of the results of the analysis and the actual measurements would compare in two orthogonal dimensions. First, it would compare the tightness of the model with respect to the actual hardware measurements. Second, it would compare the exercised path with the worst-case path. In order to fix the comparison to one degree of freedom, we compared the energy estimates obtained from static analysis and those obtained from the energy model based on the actual PMC measurements from the platforms. The tightness of the models has already been demonstrated in Section 4.

In contrast to the safety-critical embedded hard real-time software that is usually analysed with aiT, the BEEBS benchmarks also contain dynamic memory management using `malloc` and `free`. We did not analyse these benchmarks because the manual annotation effort to get tight results would be too high. Some of the benchmarks contain computed calls via function pointers that cannot be resolved automatically. In this case we manually annotated the call targets. Moreover, we specified constant data in some cases.

For the LEON3, only a subset of the BEEBS benchmarks has been measured on the hardware setup, because the execution time of some of the benchmarks is too low to synchronise the FPGA and the ASIC (see Section 4 and accompanying paper [20]).

5.1 EnergyAnalyzer for ARM Cortex-M0

For some benchmarks, the static analysis was not able to derive all loop bounds automatically. In this case, we used **Thumbulator** to derive flow constraints for the ILP-based path analysis. However, the benchmark might not exercise the worst-case path, and thus, using the simulation trace might not result in the worst-case amount of loop iterations for each loop in the program. For one of the benchmarks – *wikisort* – the simulation with **Thumbulator** fails because the binary allocated only 4096 bytes of stack, but one routine already needed 4520 bytes of stack. This causes a stack overflow. Hence, some function pointer variables are overwritten, and the benchmark cannot be executed correctly. We thus excluded the benchmark from the evaluation. Two of the benchmarks – *qsort* and *select* – contain out-of-bounds accesses.

Table 2 shows the results of the evaluation of **EnergyAnalyzer for ARM Cortex-M0**. For 43 benchmarks, the difference between the model and the static analysis is less than one percent, i.e., the execution path exercised during the simulator run is the worst-case path. Note that the analysis results include the energy consumption of the execution of the main routine, which is not included in the simulator result, which only contains the path between

the start trigger and the stop trigger. However, the contribution of this overhead is less than one mJ and hence, negligible. For the other benchmarks, the static analysis selected different paths as worst-case execution paths. The maximal observed difference between the simulator run and the static analysis is 109% for benchmark *nsichneu*, which models a state machine with many different execution paths, and the static analysis was not able to prune infeasible paths. Since the path analysis is a worst-case analysis, it maximises over the possible execution paths. Hence, the path analysis selects the worst-case combination which differs significantly from the simulated execution path.

EnergyAnalyzer allows to trade performance for precision by specifying how many calling and loop contexts should be distinguished during analysis. We used this feature to increase the analysis precision. The analysis of most benchmarks takes less than four minutes to complete, with the exception of five benchmarks (*rijndael*, *cubic*, *sqrt*, *nbody*, *picojpeg*), which took between 4 and 59 minutes.

■ **Table 2** Evaluation of the integration of the energy model for the ARM Cortex-M0 into static energy consumption analysis. For most benchmarks, the difference between the model and the static analysis is less than one percent, i.e., the execution path exercised during the simulator run is the worst-case path. For the other benchmarks, the simulated execution path and the path found by the worst-case path analysis differ significantly.

Benchmark	Analysis Result	Model Result	Δ	Note
aha-compress	78.885 mJ	78.828 mJ	< 1%	
aha-mont64	99.396 mJ	99.396 mJ	< 1%	
bubblesort	366.763 mJ	366.762 mJ	< 1%	
cnt	42.813 mJ	42.804 mJ	< 1%	
compress	27.895 mJ	27.895 mJ	< 1%	
crc	9.623 mJ	9.623 mJ	< 1%	
cubic	7.801 J	4.138 J	89%	flow constraints
duff	4.349 mJ	4.349 mJ	< 1%	
edn	302.762 mJ	302.762 mJ	< 1%	
expint	43.315 mJ	43.315 mJ	< 1%	
fac	2.934 mJ	2.904 mJ	1%	
fasta	29.383 J	21.100 J	39%	flow constraints
fdet	12.292 mJ	12.292 mJ	< 1%	
fibcall	1.493 mJ	1.493 mJ	< 1%	
fir	1.994 J	1.994 J	< 1%	
frac	1.183 J	1.183 J	< 1%	
insertsort	3.089 mJ	3.089 mJ	< 1%	
janne_complex	1.402 mJ	1.402 mJ	< 1%	
jfdctint	31.481 mJ	31.476 mJ	< 1%	
lcdnum	886.941 uJ	805.000 mJ	10%	
levenshtein	400.926 mJ	400.926 mJ	< 1%	
ludcmp	174.559 mJ	174.559 mJ	< 1%	
matmult-float	1.537 J	1.537 J	< 1%	
matmult-int	842.724 mJ	842.649 mJ	< 1%	
minver	131.316 mJ	84.348 mJ	56%	flow constraints
nbody	25.844 J	25.844 J	< 1%	
ndes	293.387 mJ	293.297 mJ	< 1%	
nettle-arcfour	105.880 mJ	105.880 mJ	< 1%	
nettle-cast128	23.214 mJ	23.211 mJ	< 1%	
nettle-des	22.595 mJ	22.595 mJ	< 1%	
nettle-md5	5.467 mJ	5.467 mJ	< 1%	
nettle-sha256	50.507 mJ	50.507 mJ	< 1%	
newlib-exp	70.439 mJ	70.439 mJ	< 1%	
newlib-log	52.954 mJ	52.954 mJ	< 1%	
newlib-sqrt	10.289 mJ	10.289 mJ	< 1%	
nsichneu	61.017 mJ	29.185 mJ	109%	
picojpeg	4.885 J	4.885 J	< 1%	
prime	209.663 mJ	209.663 mJ	< 1%	
qsort	27.294 mJ	20.408 mJ	34%	flow constraints
qurt	139.891 mJ	139.890 mJ	< 1%	
rijndael	7.176 J	7.042 J	2%	
sglib-arraybsearch	76.596 mJ	76.596 mJ	< 1%	
sglib-arrayheapsort	86.857 mJ	86.857 mJ	< 1%	
sglib-arrayquicksort	65.600 mJ	65.600 mJ	< 1%	
sglib-queue	126.250 mJ	126.250 mJ	< 1%	
shre	206.734 mJ	206.734 mJ	< 1%	
sqrt	11.529 J	11.529 J	< 1%	
st	4.142 J	2.945 J	41%	flow constraints
statemate	13.331 mJ	9.308 mJ	43%	
stb_perlin	5.145 J	5.145 J	< 1%	
stringsearch1	46.362 mJ	46.362 mJ	< 1%	
strstr	5.480 mJ	5.480 mJ	< 1%	
trio-sprintf	105.378 mJ	65.427 mJ	61%	flow constraints
trio-sscanf	139.345 mJ	71.618 mJ	95%	flow constraints
ud	21.863 mJ	21.862 mJ	< 1%	
whetstone	22.533 J	16.687 J	35%	flow constraints

5.2 EnergyAnalyzer for LEON3

Some of the BEEBS benchmarks contain floating-point computations. However, since the FPGA implementation of the LEON3 was built without a FPU, the benchmarks cannot use floating-point instructions but must use a software library that emulates these floating-point computations. One of the benchmarks – *minver* – computes a matrix multiplication using floating-point numbers, where one of the matrices is never initialised. Thus, the analysis has no knowledge about the possible floating-point values and the actual execution on the CPU will process random values, depending on what was stored in the respective memory cells. We performed both the standard worst-case analysis for this benchmark and an analysis where we assumed that the computations only process normalised IEEE 754 floating-point

numbers and zero. This reflects the “flush to zero” option present in many architectures. The computed energy consumption estimate is then cut in half, which shows that enabling “flush to zero” in software floating-point computations can save a lot of energy.

The LEON3 implements the SPARCv8 ISA, which uses register windows for fast context switches, and for providing hardware support for the call stack. However, the number of register windows is limited. The particular LEON3 model used for our experiments, available on the GR712RC board and its FPGA equivalent, has eight register windows. Due to the overlapping nature of the register windows, and their use as a ring buffer, only seven are usable. Hence, in case a program needs more than seven register windows, the processor triggers software traps to handle the register window overflow (and underflow). This happens for two of the benchmarks – *picojpeg* and *slre*. Hence, the processor needs to execute trap functions when it detects a register window overflow or a register window underflow. This causes additional energy consumption which must be taken into account during a system-level energy analysis.

Table 3 shows the results of the evaluation of **EnergyAnalyzer** for LEON3. The analysis of most benchmarks takes less than four minutes to complete, but for three benchmarks – *matmult-float*, *nbody*, and *picojpeg* – the analysis duration was 50 minutes, 45 minutes, and 24 minutes, respectively.

■ **Table 3** Evaluation of the integration of the *ISA+Cache* energy model for the LEON3 into static energy consumption analysis. For *minver*, the measured execution path and the path found by the worst-case path analysis differ significantly (see text). The costs of traps are not included in the microarchitectural analysis (see text).

Benchmark	Analysis Result	Model Result	Δ	Note
aha-compress	11.004 J	11.004 J	0%	
aha-mont64	7.499 J	7.491 J	<1%	
bubblesort	3.898 J	3.889 J	<1%	
edn	39.186 J	39.186 J	0%	
fir	159.469 J	159.469 J	0%	
frac	59.391 J	59.339 J	<1%	
levenshtein	25.506 J	25.491 J	<1%	
ludcmp	10.992 J	10.814 J	2%	
matmult-float	2.847 J	2.822 J	1%	
minver	14.372 J	4.643 J	210%	worst-case
minver	7.398 J	4.643 J	59%	assumptions
nbody	4.512 J	4.496 J	<1%	
ndes	24.828 J	24.467 J	1%	

Benchmark	Analysis Result	Model Result	Δ	Note
nettle-aes	19.401 J	19.389 J	<1%	
nettle-arcfour	9.644 J	9.639 J	<1%	
nettle-sha256	2.763 J	2.754 J	<1%	
newlib-exp	4.374 J	4.319 J	1%	
newlib-log	3.284 J	3.252 J	1%	
picojpeg	503.732 J	503.918 J	<-1%	traps
prime	3.670 J	3.667 J	<1%	
qurt	8.001 J	7.958 J	1%	
sglib-arraybinsearch	6.283 J	6.281 J	<1%	
sglib-arrayheapsort	13.066 J	13.062 J	<1%	
sglib-arrayquicksort	13.066 J	13.052 J	<1%	
sglib-queue	13.901 J	13.900 J	<1%	
slre	14.988 J	15.261 J	-2%	traps

6 Integration into TeamPlay Toolchain and Case Studies

EnergyAnalyzer for ARM Cortex-M0 and **EnergyAnalyzer** for LEON3 can be used as standalone tools to estimate the energy consumption of embedded software. They provide a rich and user-friendly graphical user interface to ease the analysis process. However, they have been developed during the TeamPlay project as part of a larger toolchain where they enable multi-criteria optimisation in a compiler, contract-based programming, and energy-aware scheduling. In the following, we present the integration of **EnergyAnalyzer** into the WCET-aware C compiler WCC [8].

The mechanisms implemented to integrate **EnergyAnalyzer** within WCC mirror the mechanisms in place to perform WCET analysis using AbsInt’s WCET analyser aiT. XTC files [10] are used to call aiT and **EnergyAnalyzer** in batch mode (i.e., without graphical user interface). An XTC file specifies the binary to be analysed, the entry point, the path to an annotation file which contains details about the target architecture configuration as well as user-provided annotations like flow facts, and the path to an XML report file. This XML

output file is then parsed by WCC after invocation of **EnergyAnalyzer** to extract the analysis results and import them into WCC's Low-Level IR at function and basic block level. This attached energy data can further be exploited by WCC to perform various compiler-level energy-aware optimisations, and thus, establishing a smooth flow between compiler-level energy analysis and optimisation. WCC supports source-level flow facts utilising ANSI C pragmas. A user can annotate their code with loop bounds, recursion depths, and execution frequency of an instruction relative to some other instruction. These source-level pragmas are translated within WCC into AIS2 annotations for aiT and **EnergyAnalyzer**.

EnergyAnalyzer has been applied to several use cases in the course of the TeamPlay project. First, it has been used to select candidates from a set of implementations of compute elements for an optimised convolutional neural network (CNN). Acting as an evaluation guide, it helped decide which optimisations should be considered for the final CNN implementation and which showed unacceptable energy consumption and should not be used. Second, it has been used to analyse the energy consumption of a camera pill prototype. The addition of an encryption algorithm showed a significant impact on the energy consumption of the camera pill that also varied depending on the type of encryption algorithm, with SPECK being an order of magnitude more energy efficient than AES and PRESENT. **EnergyAnalyzer** closely predicted the actual energy usage that was physically measured on the system with prediction relative errors ranging from 1% to 5%, depending on the encryption algorithm evaluated, and thus is a viable method for estimating energy consumption of a system such as the camera pill. Third, **EnergyAnalyzer** has been used to analyse the energy consumption of a piece of satellite software. One of the main challenges of the space industry is power consumption, as spacecrafts usually have limited access to power sources. **EnergyAnalyzer** proved to be a useful tool thanks to its support of the LEON3 on the GR712RC platform, which is the most common processor ASIC used by European space companies. The results showed a precise prediction of the energy consumption for the different binaries, with <1% estimation error compared with physical measurements for the final optimised version compiled with WCC. Another interesting feature of **EnergyAnalyzer** was the result visualisation using call and control-flow graphs, showing the energy consumption for each of the functions inside the binary which could be used not only for predicting and minimising energy consumption but also for qualifying code for space.

More details on the integration of **EnergyAnalyzer** in the TeamPlay toolchain and the use cases on which the toolchain has been applied are presented in Rouxel et al. [26].

7 Conclusion

This paper presents our work on static energy consumption analysis for embedded systems. We created energy models for two predictable architectures: the ARM Cortex-M0 and the Gaisler LEON3, both achieving estimation errors of less than 10% when validated against our two target hardware platforms. Both models are based on hardware event counters, which can be predicted by static analysis. The models are integrated into **EnergyAnalyzer**, a novel tool for code-level static energy consumption analysis. Our evaluation results show a good accuracy of the energy consumption analysis. The tool provides a user-friendly graphical interface to analyse energy consumption at different levels of granularity, from the entire program to individual functions and basic blocks. **EnergyAnalyzer** is part of the TeamPlay toolchain [26], where it enables multi-criteria code optimisation during compilation.

We demonstrated the usefulness of our tools in several case studies including a camera-pill designed for medical diagnosis and a space communications platform. In each case, the tool provided precise predictions of the energy consumption and helped to identify

energy bottlenecks. In conclusion, our work provides a useful approach to analyse energy consumption in embedded systems. A potential avenue for extension is to include peripheral energy consumption for system-level analysis [31]. We believe that our approach can help to design more energy-efficient embedded systems and applications, which is a crucial step towards sustainable development.

References

- 1 ARM Ltd. The ARM Cortex-M0 processor. <https://developer.arm.com/products/processors/cortex-m/cortex-m0>. Accessed: 2018-08-14.
- 2 Cobham Gaisler AB. GR712RC Dual-Core LEON3-FT Development board. <https://www.gaisler.com/index.php/products/boards/gr712rc-board>. Accessed: 2018-09-11.
- 3 Cobham Gaisler AB. LEON3FT Fault-tolerant processor. <https://www.gaisler.com/index.php/products/processors/leon3ft>. Accessed: 2018-09-11.
- 4 Cobham Gaisler AB. GRLIB IP Core User's Manual, 2019. URL: <https://www.gaisler.com/products/grlib/grip.pdf>.
- 5 Martin Cochet, Guillaume Bonnechere, Jean-Marc Daveau, Fady Abouzeid, and Philippe Roche. Implementing the LEON3 Statistics Unit in 28nm FD-SOI: Power Estimation by Activity Proxy, 2016.
- 6 Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2003. URL: <http://nbn-resolving.de/urn:nbn:se:uu:diva-3502>.
- 7 Zbigniew Chamski et al. The Thumbulator Git repository modified for collecting performance monitoring counters for the STM32F0-Discovery board. Branch: prefetch-model, tag: teamplay-D4.5. URL: <https://github.com/PicoPET/thumbulator-stm32f0x.git>.
- 8 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010.
- 9 Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. Energy transparency for deeply embedded programs. *ACM Trans. Archit. Code Optim.*, 14(1), March 2017. doi:10.1145/3046679.
- 10 AbsInt Angewandte Informatik GmbH. XTC Language Specification, Version 2.7. <https://www.absint.com/xtc/xtc-specification.pdf>. Accessed: 2020-01-13.
- 11 Xinfei Guo, Vaibhav Verma, Patricia Gonzalez-Guerrero, and Mircea R Stan. When “things” get older: exploring circuit aging in iot applications. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 296–301. IEEE, 2018.
- 12 Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings*, pages 288–297. IEEE Computer Society, 1995. doi:10.1109/REAL.1995.495218.
- 13 Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 81–90. IEEE Computer Society, 2006. doi:10.1109/RTAS.2006.17.
- 14 Michael H. Kutner, Chris Nachtsheim, John Neter, and William Li. *Applied linear statistical models*. McGraw-Hill Irwin, 2005.
- 15 Charles L. Lawson and Richard J. Hanson. *Solving least squares problems*. SIAM, 1995.
- 16 Jeremy Morse, Steve Kerrison, and Kerstin Eder. On the limitations of analyzing worst-case dynamic energy of processing. *ACM Trans. Embed. Comput. Syst.*, 17(3):59:1–59:22, February 2018. doi:10.1145/3173042.
- 17 Krastin Nikov and José L. Núñez-Yáñez. Intra and inter-core power modelling for single-isa heterogeneous processors. *Int. J. Embed. Syst.*, 12(3):324–340, 2020. doi:10.1504/IJES.2020.107046.

- 18 Krastin Nikov, José L. Núñez-Yáñez, and Matthew Horsnell. Evaluation of hybrid run-time power models for the ARM big.little architecture. In Eli Bozorgzadeh, João M. P. Cardoso, Rui Abreu, and Seda Ogrenci Memik, editors, *13th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013, Porto, Portugal, October 21-23, 2013*, pages 205–210. IEEE Computer Society, 2015. doi:10.1109/EUC.2015.32.
- 19 Kris Nikov, Kyriakos Georgiou, Zbigniew Chamski, Kerstin Eder, and José L. Núñez-Yáñez. Accurate energy modelling on the cortex-m0 processor for profiling and static analysis. In *29th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2022, Glasgow, United Kingdom, October 24-26, 2022*, pages 1–4. IEEE, 2022. doi:10.1109/ICECS202256217.2022.9971086.
- 20 Kris Nikov, Marcos Martínez, Simon Wegener, José L. Núñez-Yáñez, Zbigniew Chamski, Kyriakos Georgiou, and Kerstin Eder. Robust and accurate fine-grain power models for embedded systems with no on-chip PMU. *IEEE Embed. Syst. Lett.*, 14(3):147–150, 2022. doi:10.1109/LES.2022.3147308.
- 21 Jose Nunez-Yanez and Geza Lore. Enabling accurate modeling of power and energy consumption in an ARM-based System-on-Chip. *Microprocessors and Microsystems*, 37(3):319–332, 2013.
- 22 James Pallister, Simon J. Hollis, and Jeremy Bennett. BEEBS: open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013. arXiv:1308.5174.
- 23 James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPEs 2017*, pages 51–59, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078659.3078666.
- 24 Santhosh Kumar Rethinagiri, Oscar Palomar, Rabie Ben Atitallah, Smail Niar, Osman Unsal, and Adrian Cristal Kestelman. System-level power estimation tool for embedded processor based platforms. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1–8, 2014.
- 25 Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. A study on the use of performance counters to estimate power in microprocessors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(12):882–886, 2013.
- 26 Benjamin Rouxel, Christopher Brown, Emad Ebeid, Kerstin Eder, Heiko Falk, Clemens Grellck, Jesper Holst, Shashank Jadhav, Yoann Marquer, Marcos Martinez de Alejandro, Kris Nikov, Ali Sahafi, Ulrik Pagh Schultz Lundquist, Adam Seewald, Vangelis Vassalos, Simon Wegener, and Olivier Zendra. The teamplay project: Analysing and optimising time, energy, and security for cyber-physical systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023. doi:10.23919/DATE56975.2023.10137198.
- 27 Adam Seewald, Ulrik Pagh Schultz, Emad Ebeid, and Henrik Skov Midtiby. Coarse-grained computation-oriented energy modeling for heterogeneous parallel embedded systems. *International Journal of Parallel Programming*, 49(2):136–157, 2021.
- 28 Volkmar Sieh, Robert Burlacu, Timo Hönig, Heiko Janker, Phillip Raffeck, Peter Wägemann, and Wolfgang Schröder-Preikschat. An end-to-end toolchain: From automated cost modeling to static WCET and WCEC analysis. In *20th IEEE International Symposium on Real-Time Distributed Computing, ISORC 2017, Toronto, ON, Canada, May 16-18, 2017*, pages 158–167. IEEE Computer Society, 2017. doi:10.1109/ISORC.2017.10.
- 29 STMicroelectronics. STM32F030x4/x6/x8/xC and STM32F070x6/xB advanced ARM-based 32-bit MCUs - Reference Manual. Accessed: 2020-12-28. URL: https://www.st.com/resource/en/reference_manual/dm00091010-stm32f030x4x6x8xc-and-stm32f070x6xb-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.
- 30 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000. doi:10.1023/A:1008141130870.

- 31 Peter Wagemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPICs*, pages 24:1–24:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECRTS.2018.24.
- 32 Peter Wagemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 105–114. IEEE Computer Society, 2015. doi:10.1109/ECRTS.2015.17.
- 33 Matthew J Walker, Stephan Diestelhorst, Andreas Hansson, Anup K Das, Sheng Yang, Bashir M Al-hashimi, and Geoff V Merrett. Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs. *Ieee Transactions on Computer Aided Design of Integrated Circuits and Systems*, 36(1):1–14, 2017. doi:10.5258/SOTON/393673.
- 34 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008. doi:10.1145/1347375.1347389.

A LEON3 Energy Model Selection

■ **Table 4** All supported PMCs by EnergyAnalyzer.

#	Counter	#	Counter	#	Counter
C_0	TIME	C_5	WBHOLD	C_{14}	LDST
C_1	ICMISS	C_7	IINST	C_{15}	LOAD
C_2	ICHOLD	C_{11}	BRANCH	C_{16}	STORE
C_3	DCMISS	C_{12}	CALL		
C_4	DCHOLD	C_{13}	TYPE2		

■ **Table 5** Coarse-grained Model Results for the Gaisler GR712RC platform.

Model	Expression	MAPE[%]	
		Train	Test
Energy [J] All Supported All Events	$E = 0.155261 + 2.94155e-08 \times C_0$ $+ 2.5661e-09 \times C_2 + 9.93453e-09 \times C_5$ $+ 8.97535e-10 \times C_{12} + 3.21255e-09 \times C_{13}$ $+ 6.14384e-09 \times C_{15} + 4.54827e-08 \times C_{16}$	1.14	0.29
Energy [J] All Supported Bottom-Up	$E = 0 + 3.19557e-08 \times C_0$ $+ 5.79224e-08 \times C_{16}$	1.20	1.38
Energy [J] All Supported Top-Down	$E = 0.131077 + 3.13122e-08 \times C_0$ $+ 9.17778e-09 \times C_5 + 2.99043e-09 \times C_{15}$ $+ 3.92999e-08 \times C_{16}$	1.02	1.54
Energy [J] All Supported Full-Exhaustive	$E = 0.131087 + 3.13122e-08 \times C_0$ $+ 9.17779e-09 \times C_5 + 2.99043e-09 \times C_{14}$ $+ 3.63095e-08 \times C_{16}$	1.02	1.54
Energy [J] IsaCache All Events	$E = 0 + 1.18567e-06 \times C_3$ $+ 5.9072e-07 \times C_{12} + 3.88949e-08 \times C_{13}$ $+ 8.03337e-08 \times C_{14} + 6.89885e-08 \times C_{16}$	8.38	24.03
Energy [J] IsaCache Bottom-Up	$E = 0 + 3.93365e-08 \times C_7$ $+ 1.87111e-07 \times C_{16}$	5.84	8.24
Energy [J] IsaCache Top-Down	$E = 0 + 3.93365e-08 \times C_7$ $+ 1.87111e-07 \times C_{16}$	5.84	8.24
Energy [J] IsaCache Full-Exhaustive	$E = 0 + 3.93365e-08 \times C_7$ $+ 1.87111e-07 \times C_{16}$	5.84	8.24

We have chosen the *ISA+Cache* subset of PMCs shown in Table 1 because these events can be statically predicted with highest accuracy. However, there are more events that are statically predictable. A list of all the supported PMCs can be found in Table 4, with further information available in the L3STAT User Manual [4]. Separate models are generated using each of the PMC subsets. In addition to using the *bottom-up* and *top-down* search algorithms, detailed in [20], the relatively small PMC sets also allow for a *full-exhaustive* search to be used. The resulting models are then compared against a model that uses all available PMCs. Table 5 presents the results of the *coarse-grained* model generation and

evaluation. The `train` and `test` benchmark sets used are the same as the ones used for the fine-grain model generation and validation, presented in [20]. As expected, the models computed using the larger *All Supported* PMC list perform better than the ones using the *ISA+Cache* list. However, it is interesting to note that the three search algorithms exploring the *ISA+Cache* PMCs all converge on the same model with the *STORE* event present in all models generated, regardless of PMC selection and search method. The reason why the *ISA+Cache* models perform worse than the *All Supported* models is that the *TIME* event, which is the single best predictor of power/energy according to previous work [18, 17], is not included in the list. However, the reduced precision during microarchitectural analysis for the *All Supported* subset outweigh the higher model accuracy. Additionally, it seems that if only one search algorithm can be used due to time limitation, the *top-down* search seems to produce overall better models than *bottom-up* and very similar models to *full-exhaustive* while taking only a fraction of the time to search through the list of events.