17th International Workshop on Worst-Case Execution Time Analysis

WCET 2017, June 27, 2017, Dubrovnik, Croatia

Edited by

Jan Reineke

Editor

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken
Germany
reineke@cs.uni-saarland.de

ACM Classification 1998
B.8.2 Performance Analysis and Design Aids, C.3 Real-Time and Embedded Systems, D.2.4 Software/ Program Verification, D.4.7 [Organization and Design] Real-Time Systems and Embedded Systems

ISBN 978-3-95977-057-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-057-6.

Publication date
June, 2017

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

Digital Object Identifier: 10.4230/OASIcs.WCET.2017.0

ISBN 978-3-95977-057-6 **ISSN 1868-8969** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

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

ISSN 2190-6807

http://www.dagstuhl.de/oasics

Contents

Regular Papers

17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017).
Editor: Jan Reineke
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

It is my pleasure to welcome you to the proceedings of the *17th International Workshop on Worst-Case Execution Time Analysis* (WCET 2017). This year the proceedings are published prior to the workshop in order to further stimulate interaction among participants at the event. WCET 2017 is going to take place in Dubrovnik, Croatia as a satellite event of the *29th Euromicro Conference on Real-Time Systems* (ECRTS 2017), the premier European venue for research in the broad area of real-time and embedded systems.

The goal of the workshop is to bring together people from academia, tool vendors and users in industry who are interested in all aspects of timing predictability of real-time systems. This is reflected by the makeup of the program committee, as well as the authors of the papers in the proceedings, which consists of academic researchers, tool vendors, and colleagues from industry. This year we received 14 high-quality submissions, out of which the program committee selected 10 for presentation at the workshop and for publication in the proceedings. These papers cover a broad range of timely topics, including, among others, parallel real-time benchmarks, early-stage WCET prediction using machine learning, and approaches to multi-core WCET analysis. I hope that you will find the program interesting and inspiring for your future work!

Putting together this year's workshop program was a team effort. First of all, I would like to thank the authors for their submissions. The program committee and the external reviewers did a great job in evaluating the submissions and in providing constructive feedback to the authors. Thank you for that! I am also grateful to the ECRTS 2017 general chair, Martina Maggio and her team, and the Real-Time Technical Committee Chair of Euromicro, Gerhard Fohler, for their support in organizing the workshop. Thank you also to the team at Schloss Dagstuhl, in particular Marc Herbstritt, for their help in preparing these proceedings.

I am looking forward to welcoming you to WCET 2017 in Dubrovnik! I encourage all participants to embrace the opportunity for discussion and interaction with the authors and other workshop attendees.

Saarbrücken, May 26, 2017
Jan Reineke

17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017).
Editor: Jan Reineke

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

Committee

Program Chair

- Jan Reineke – Saarland University, Germany

Program Committee

- Sebastian Altmeyer – University of Amsterdam, Netherlands
- Clément Ballabriga – Lille 1 University, France
- Florian Brandner – Télécom ParisTech, France
- Hugues Cassé – IRIT - Université de Toulouse, France
- Francisco J. Cazorla – Barcelona Supercomputing Center, Spain
- Heiko Falk – Hamburg University of Technology, Germany
- Niklas Holsti – Tidorum Ltd, Finland
- Claire Maïza – Grenoble INP/Verimag, France
- Kartik Nagar – Purdue University, United States
- Luis Miguel Pinho – CISTER Research Centre/ISEP, Portugal
- Dumitru Potop Butucaru – INRIA Rocquencourt, France
- Wolfgang Puffitsch – Oticon A/S, Denmark
- Peter Puschner – Vienna University of Technology, Austria
- Martin Schoeberl – Technical University of Denmark, Denmark
- Benoît Triquet – Airbus Group, France

External Reviewers

- Pedro Benedicte – Barcelona Supercomputing Center, Spain
- Sebastian Hahn – Saarland University, Germany
- Farouk Hebbache – CEA LIST, France
- Michael Jacobs – Saarland University, Germany
- Suzana Milutinovic – Barcelona Supercomputing Center, Spain
- Amine Naji – ENSTA ParisTech, France
- Dominic Oehlert – Hamburg University of Technology, Germany

Steering Committee

- Guillem Bernat – Rapita Systems Ltd., United Kingdom
- Björn Lisper – Mälardalen University, Sweden
- Isabelle Puaut – University of Rennes I/IRISA, France
- Peter Puschner – Vienna University of Technology, Austria

17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017).
Editor: Jan Reineke

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

STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation & Real-Time Scheduling

Benjamin Rouxel[1] **and Isabelle Puaut**[2]

1 **University of Rennes 1, Rennes, France**
`benjamin.rouxel@irisa.fr`
2 **University of Rennes 1, Rennes, France**
`isabelle.puaut@irisa.fr`

Abstract

We all had quite a time to find non-proprietary architecture-independent exploitable parallel benchmarks for Worst-Case Execution Time (WCET) estimation and real-time scheduling. However, there is no consensus on a parallel benchmark suite, when compared to the single-core era and the Mälardalen benchmark suite [11]. This document bridges part of this gap, by presenting a collection of benchmarks with the following good properties: (i) easily analyzable by static WCET estimation tools (written in structured C language, in particular neither *goto* nor dynamic memory allocation, containing flow information such as loop bounds); (ii) independent from any particular run-time system (MPI, OpenMP) or real-time operating system. Each benchmark is composed of the C source code of its tasks, and an XML description describing the structure of the application (tasks and amount of data exchanged between them when applicable). Each benchmark can be integrated in a full end-to-end empirical method validation protocol on multi-core architecture. This proposed collection of benchmarks is derived from the well known StreamIT [21] benchmark suite and will be integrated in the TACleBench suite [10] in a near future. All these benchmarks are available at `https://gitlab.inria.fr/brouxel/STR2RTS`.

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

Keywords and phrases Parallel benchmarks, Tasks scheduling, Worst-Case Execution Time estimation

Digital Object Identifier 10.4230/OASIcs.WCET.2017.1

1 Motivations

In the past, the benchmark suite provided by the Mälardalen institute [11] has been widely accepted by the community studying the Worst-Case Execution Time (WCET) of real-time applications on single-core architectures. While multi-cores tend to replace mono-core architectures, no consensus emerged on a parallel benchmark suite when studying the Worst-Case Response Time (WCRT) of a parallel application or its global WCET. The main unsatisfied requirement of such a benchmark suite lies on the identification of parallel tasks to benefit from the multiplicity of available cores.

Current research papers on real-time multi-core mapping and scheduling already face this issue and already use representative application codes for validation. However, it is a common practice to use proprietary applications from the automotive or avionic world [1, 18], unfortunately preventing other researchers to replay tests or to compare results.

Several non proprietary parallel benchmarks already exist for the experimental validation of real time systems. However, they have some limitations. Some consist of periodical

independent task sets [4, 6] only, with no synchronization/communication between tasks. Some others lack information to perform WCET estimation or scheduling, e.g.: source code [8, 9, 20], dependency representation [16, 13, 2, 19, 12], or are hardware or run-time system dependent. Other studies prefer task set generators for validation, but cannot be used for a full end-to-end experimental validation as they lack source code.

This document aims at providing a collection of parallel benchmarks for experimental evaluation of real-time systems on multi-/many-core architectures. The targeted audience is the real-time system research community at large, including researchers on WCET estimation and real-time scheduling. This document can be of benefit to experts in multi-core scheduling to experiment their techniques for task mapping and scheduling. It can be of benefit to researchers on worst-case execution time estimation, both on single-core architectures, by analyzing each task of the parallel application, and on multi-core architectures through an analysis of the entire parallel application, including for instance analyses of contentions at the shared resources such as bus, cache, etc.

To ease the creation of a collection of benchmarks with all the required information for WCET estimation and scheduling, we started from the StreamIT benchmark suite [21], which consists of a set of Digital Signal Processing (DSP) applications. Such applications consume incoming data and produce outgoing data at a specific rate, which is representative of many real-time applications.

The provided information for each application is an XML file and a C source file. The XML file describes the structure of the application through a directed acyclic graph (identification of tasks and dependencies between them, volume of data to be transmitted between tasks, WCET of tasks on a particular architecture if the benchmark is to be used for real-time scheduling only). The C code contains the source code of each task. The source code is statically analyzable and self-contained, to allow static WCET estimation techniques on any specific architecture, (but obviously other estimation techniques such as probabilistic or measurement-based are not left aside). In particular, the C code contains pragmas expressing loop bounds in the format used in the TACleBench benchmark suite [10]. We plan to integrate them in the TACleBench benchmark suite as it aims to be the reference benchmark suite for WCET estimation at code level for both single-core and multi-/many-core architecture.

The rest of this document is organized as follows. First, Section 2 compares our work with existing benchmark suites. Section 3 presents background knowledge about the StreamIT benchmark suite, which is used as the basis for STR2RTS. Section 4 provides an overview of the provided material, and finally Section 5 gives some qualitative and quantitative information on the provided benchmarks, before concluding in Section 6.

2 Related Work

The usefulness of benchmarks for the validation of systems no longer has to be demonstrated. They have been vastly used in the past to experiment new algorithms, new software, or new pieces of hardware. In computer science, there exists hundreds of different benchmark suites with different purposes and different sizes: SPEC CPU 2006, PolyBench [17], ParMiBench [12], UTDSP [13], Parsec [2], JemBench [19], ParaSuite [16], and many more. However very few of them have been engineered for *multi-core real-time* systems. This kind of systems requires more information in the benchmark suite than just the code, typical input data and a description, which is the general provided material. Indeed, to be largely accepted by the real-time community a benchmark suite must include a source code that is statically analyzable, to allow experiments with both static and non-static WCET estimation methods.

A non-exhaustive list of the requirements for benchmarks targeting real-time embedded systems would include (i) structured self-contained source code – i.e.: no goto, no dynamic memory allocation, no call to external libraries, (ii) statically computable loop bounds or flow facts for loop bounds, (iii) deadlines and periods of tasks. Adding the multi-core constraints to the system would also add new requirements to the benchmark suite, such as the amount of data exchanged between communicating tasks, and a representation of dependencies between tasks if applicable. The benchmark suite should also remain independent from any specific run-time environment (i.e.: OpenMP, MPI, etc.) to be used as easily as possible.

Starting from the single-core era, the Mälardalen benchmark suite [11] has been accepted by the WCET estimation community. It consists of small pieces of key code representing some well-known code structures found in embedded real-time software. Although representative of embedded software, this benchmark suite contains sequential codes only, and the large majority of provided codes are very small.

A common practice to evaluate scheduling strategies is to use task graph generators. They have the benefit to be architecture independent and generate a vast amount of different topologies. Task Graph For Free (TGFF) [9], Synchronous Dataflow 3 (SDF3) [20] can generate task graphs with dependent tasks in a deterministic way, allowing anybody to replay an experiment as long as the configuration parameters are known. UUniFast [4] is an algorithm generating task sets with uniform distribution in a given space. Task graph generators are very useful when the goal is to empirically validate a method on a large variety of task graph topologies. However we all need *concrete* representative applications with code for further empirical validation which is what we aim at providing here.

Three real parallel applications targeting real-time systems are often used as benchmarks – i.e.: Debie1 [7], Papabench [14] and Rosace [15]. All are control applications respectively for a satellite, a drone and a plane. But three concrete applications are not enough. Our objective with the benchmark suite we provide is to enrich the set of applications that can be used to validate multi-core real-time systems, and enlarge the scope of applications to include signal processing applications with dependencies between tasks.

De Bock *et al.* [6] proposed a benchmark generator targeting multi-core platforms. The generator input is sequential code for each task. All tasks are independent. The benchmark generator output is a task-set fitting some requirements. In comparison, the benchmarks we provide include *dependent* tasks and a representation of these dependencies as well as the amount of data exchanged between dependent tasks.

To the best of our knowledge the benchmark suite closest to this work is the StreamIT benchmark suite [21] that we use as a baseline. In the original version of StreamIT, the authors provide a representation of task's dependencies – a task graph – with communication (exchanged tokens) and source code. However, the provided C source code is only sequential, and the generated code is not WCET-friendly as some benchmarks are impossible to statically analyze, i.e. statically extracting loop bounds might not be possible with available tools. In addition, there is nearly no cache reuse, since tasks performing the same function are systematically duplicated in the generated code. Moreover, dynamic memory allocation is used for allocating messages used for inter-task communication. The C version we provide respects the task graph extracted from the original StreamIT tool, with the benefit of allowing static analyses on each function in isolation.

Finally the new TACleBench suite [10] aims at becoming the *de facto* standard benchmark suite for timing analysis. This work will be integrated in TACleBench in order to strengthen the multi-/many-core dimension of this suite.

```
1  void->void pipeline FFT4 {
2    add OneSource(); add FFTKernel(2);
3    add FloatPrinter();
4  }
5  float->float pipeline FFTKernel (int N) {
6    for (int i=1; i<N; i*=2) {
7      add Butterfly(i, N);
8    }
9  }
10 float->float pipeline Butterfly (int N, int
       W){
11   add splitjoin {
12     split roundrobin(N, N);
13     add Identity<float>(); add Multiply();
14     join roundrobin();
15   };
16   add splitjoin {
17     split duplicate;
18     add Add(); add Subtract();
19     join roundrobin(N, N);
20   };
21 }
22 void->float filter OneSource {...}
23 float->float filter Multiply {...}
24 float->float filter Add {
25   init {}
26   work push 1 pop 2 {
27     push(pop() + pop());
28   }
29 }
30 float->float filter Subtract {...}
31 float->void filter FloatPrinter {...}
```

Listing 1 FFT4 stream program's structure.

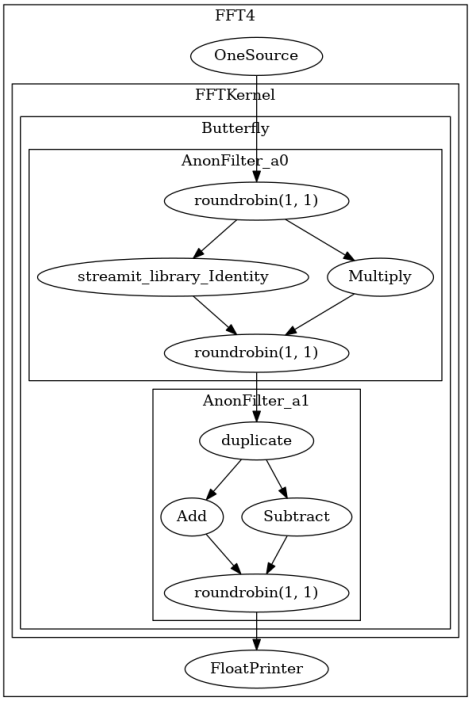


Figure 1 FFT4 SDF graph.

3 Background on StreamIT

StreamIT [21] is a high-level language for developing *streaming* applications (applications acting on flows of data) modeled as Synchronous Dataflow Graphs (SDF). The StreamIT language has a portable run-time environment and is architecture-independent. The main difference of StreamIT as compared with other streaming languages lies on a required well-defined structure on the streams, that are not an arbitrary network of nodes. One of the major properties of the StreamIT benchmarks lies on the data rate which is imposed to be fixed, thus known at compile time.

All graphs in the StreamIT languages consist of a hierarchical composition of nodes structured in *pipeline*, *split-join* and *feedbackloop* constructs. Streaming applications can then be represented as a Cyclo Static DataFlow graph (CSDF) [3]. They are constructed over the execution of two phases : the *initialization* and the *steady state*, where the latter is considered indefinitely repeating, whereas the former is performed only once and aims at registering the tasks of the steady-state in the StreamIT scheduler.

A streaming application can be seen as a flow of computational units producing and consuming data, the data stream. The basic computational unit of StreamIT applications is the *filter*. Each filter is a task that produces and consumes tokens. Communicating filters are organized in a stream in order to create a *pipeline* (chain) of *filters*. More complex stream structures can be realized with *split-join* and *feedbackloop* constructs. The former splits the data stream in parallel streams before joining them again, whereas the latter re-injects upstream data produced downstream. Conditional control-flow is not allowed at the application level (there is no concept of conditional execution of filters). In contrast, there may be control flow inside filter code. The data stream is propagated through the

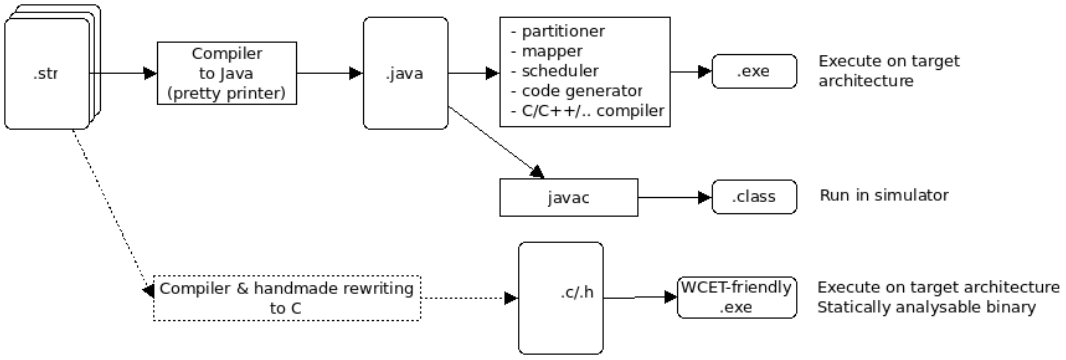


Figure 2 StreamIT Tool-chain.

filters in the graph at a constant rate known, at compile time. This allows to statically know the amount of data exchanged between filters. Such data are transmitted through dataflow channels implemented as FIFO (First In First Out) queues.

The StreamIT language is illustrated on one of the smallest application from the StreamIT benchmark suite: the radix-2 case of a Fast Fournier Transform (FFT4.str). The application source code in StreamIT language is presented in Listing 1. Lines 1–21 specify the structure of the streaming applications, while lines 22–31 give the source code of the filters (for conciseness only the StreamIT code for filter *Add* is given).

The first element (line 1: FFT4) is the top-level envelope (equivalent to the *main* function in C code); it registers three other elements which are added to the global structure (a *pipeline*, i.e. chain of elements for FFT4). Elements are added directly or recursively explored depending of their type. For instance, *OneSource*, line 22 is added directly because it is a simple filter, whereas *Butterfly*, line 10 is explored because it is a composition of elements (here, a pipeline). The code of a very simple filter (*Add*) is given in lines 24–29. It is decomposed into two functions : the *initialization* part (line 25) and the *work* function for the steady-state (lines 26–28). Due to the simplistic nature of this example the initialization part is empty, but one could easily imagine some constant initialization for the steady-state. The *work* function corresponds to the C-like code that will be executed at each iteration of the steady-state. This function calls, at line 27, two functions *pop*/*push* to respectively fetch and store data from/to the FIFO channels connected to the previous/next dependent tasks.

The program structure extracted from the StreamIT application from Listing 1 is presented in Figure 1 and it illustrates the steady-state of the application.

The StreamIT benchmark suite comes with an end-to-end compilation tool chain illustrated in Figure 2. It first parses the StreamIT language and generates a Java version of the streaming application. This Java version is then converted to an intermediate representation used by internal tools to analyze the application. Following is a short summary of those tools:

- **Partitioning:** determining the number of fissions and fusions, used to determine where to insert/remove split-join nodes in the generated code;
- **Mapping:** determining on which core each job implementing a filter will run;
- **Scheduling:** determining in which order jobs will be executed;
- **Code generation:** generating code for the targeted architecture (generally C/C++) through the provision of several back-ends.

The last step generates a code which can be compiled in order to run the application on the targeted architecture (RAW processor, Tilera, RStream and so on). The Java version can

```
<?xml version="1.0" encoding="UTF-8" ?>
<appl>
  <tasks>
  <task id="fft4_split2_duplicate" WCET="2286">
    <prev id="fft4_join1_round_robin" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_join2_weighted_round_robin" WCET="1380">
    <prev id="fft4_add" data-sent="1" data-type="float" />
    <prev id="fft4_subtract" data-sent="1" data-type="float" />
  </task>
  <task id="fft4_add" WCET="1600">
    <prev id="fft4_split2_duplicate" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_subtract" WCET="1600">
    <prev id="fft4_split2_duplicate" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_float_printer" WCET="1614">
    <prev id="fft4_join2_weighted_round_robin" data-sent="2" data-type...
  </task>
  <task id="fft4_one_source" WCET="1198"></task>
  <task id="fft4_split1_weighted_round_robin" WCET="1380">
    <prev id="fft4_one_source" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_join1_round_robin" WCET="1198">
    <prev id="fft4_identity" data-sent="1" data-type="float" />
    <prev id="fft4_multiply" data-sent="1" data-type="float" />
  </task>
  <task id="fft4_identity" WCET="1054">
    <prev id="fft4_split1_weighted_round_robin" data-sent="1" data-type...
  </task>
  <task id="fft4_multiply" WCET="1070">
    <prev id="fft4_split1_weighted_round_robin" data-sent="1" data-type...
  </task>
</tasks>
</appl>
```

Listing 2 XML representation of the FFT4 application.

also be executed using a simulation library included in the StreamIT project. This simulator runs a sequential version of the streaming application.

Despite the work done on the StreamIT toolchain, none of the provided back-ends generate code ready to be analyzed in the context of real-time systems. The *simpleC* back-end generates only one big main function containing all the code, leading to a sequential version not suitable for multi-core analysis/execution. The *newSimple* back-end generates hard to read and to analyze source code, where it is not possible anymore to identify tasks. The *cluster* back-end generates code that may not be analyzable by loop bound extractors, and includes libraries provided by StreamIT with C++ classes and dynamic memory allocation, thus not suitable for static WCET analysis. In addition, when the same filter is used several times, its code is duplicated, thus degrading the WCET of tasks when considering architectures with caches.

As no back-end fulfills all the requirements implied by real-time systems and corresponding analyses, we modified the StreamIT benchmarks code, as detailed in Section 4.2, to fit the needs of the real-time system community. Among the tools coming with the StreamIT tool chain, we only used the simulation library to ensure that our modifications to the StreamIT codes are functionally equivalent to the original code.

4 Benchmarks overview

This section presents an example of the provided information: (i) an XML description, (ii) a C source code ; and how to use it. Then, it presents how this information was extracted from the StreamIT benchmark suite [21].

4.1 Provided information

Each benchmark is divided in 4 files, an XML file, a DOT file, a C source file and its corresponding header file. The DOT file is a graphical representation of the tasks and their dependencies using the *graphviz* software[1], and will not be presented here, as well as the header file. Following is an example of an XML description with its corresponding C source code.

An XML file summarizing all the provided information is presented by Listing 2 and corresponds to our previous example from Figure 1. This file basically describes the structure of the application as a Directed Acyclic Graph (DAG), with tasks as nodes and channels as edges. It can be used by mapping/scheduling tools as input to experimentally evaluate new mapping/scheduling strategies involving either a single application or multiple applications both modeled as DAGs. Another usage, once tasks have been assigned to cores in a multi-core platform, is to use the XML file together with the code of tasks to perform WCET estimation on the application, in particular integrating contentions to access shared resources in the WCET of the application. For each task, the XML file contains the set of predecessors of the task with the amount of data received by each of them, as well as the task WCET. The XML tag *prev* represents task's dependencies as precedences, e.g. line 15 where *Split2DUPLICATE* is a predecessor of *Subtract*. The associated attribute *data-sent* specifies the amount of data needed for one execution of the task, and the attribute *data-type* specified the type of data (e.g. : int, double, float, etc.).

The attribute *WCET* is provided as information for people aiming at performing experiments on mapping/scheduling techniques and do not wishing to perform an initial WCET analysis step. Provided WCETs were estimated by our static WCET analysis tool Heptane [5] for a the MIPS instruction set, for an architecture without caches or pipelines (roughly the provided WCET corresponds to the worst-case number of instructions executed by each task).

Listing 3 introduces the structure of the provided C source code. Each task from the aforementioned graph appears as a C function in the source file. The code of filter/task *Add* is given as an example in lines 8–14. Depending on the value of $GLOBAL\_N$ (constant evaluated by the C pre-processor), this filter reads two float items from the input channel (pop_float), then sums them before writing the result into the output channel ($push\_float$). The loop is annotated with a *pragma* specifying the loop bound, according to the TACleBench syntax for flow-facts annotations. The value of $GLOBAL\_N$ has an impact on the number of added tasks (number of added *Butterfly* from the Listing 1). In the C source code, we fix such parameter in the header file to have the C source code consistent with the XML description. In this example the value of $GLOBAL\_N$ is set to 2.

Lines 22–41 point to the *sequential_main* function that corresponds to an execution of all tasks on a single-core architecture. Function *sequential_main* first calls the initialization function of all tasks having a non-empty initialization phase (line 23). This initialization step sets up every C structure, buffers or pre-computed data required by filters for the steady-state run. The *sequential_main* then calls each filter function in a loop of $MAX\_ITERATION$ iterations (lines 28–39) for the steady-state execution. Functions are called in an order that respects dependencies between tasks. This function is provided for users interested in single-core WCET estimation. It was also used to check the correctness of code modifications applied to the StreamIT benchmark, by comparing the results to those produced by the StreamIT Java simulator.

[1] `http://www.graphviz.org/`

```
#include "FFT4.h"
// GLOBAL_N is defined in the header file and its value is 2

void fft4_one_source() { ... }
void fft4_identity() { ... }
void fft4_multiply() { ... }
void fft4_add() {
_Pragma("loopbound min "GLOBAL_N/2" max "GLOBAL_N/2)
  for(int i=0 ; i < GLOBAL_N/2 ; i++) {
    float v1 = pop_float(&AddBuf.buffer_in);
    float v2 = pop_float(&AddBuf.buffer_in);
    push_float(&AddBuf.buffer_out, v1+v2);
  }
}
void fft4_subtract() { ... }
void fft4_float_printer() { ... }
void fft4_init() { ... }
void fft4_split1_weighted_round_robin( uint32_t nb ) { ... }
void fft4_join1_round_robin() { ... }
void fft4_split2_duplicate() { ... }
void fft4_join2_weighted_round_robin( uint32_t nb ) { ... }
int sequential_main( int argv, char **argc ) {
  fft4_init();
_Pragma("loopbound min "MAX_ITERATION" max "MAX_ITERATION)
  for( int i=0 ; i < MAX_ITERATION ; i++ ) {
    fft4_OneSource();
_Pragma("loopbound min "(GLOBAL_N/2-1)" max "(GLOBAL_N/2-1))
    for( int j = 1 ; j < GLOBAL_N ; j *= 2 ) {
      fft4_split1_weighted_round_robinv(j);
        fft4_identity();
        fft4_multiply();
      fft4_join1_round_robin();
      fft4_split2_duplicate();
        fft4_add();
        fft4_subtract();
      fft4_join2_weighted_round_robin(j);
    }
    fft4_float_printer();
  }
  return EXIT_SUCCESS;
}
```

Listing 3 C version of the FFT4 stream program.

Regarding communications between tasks, a C file implementing the push/pop communication functions has to be provided and linked with the code of each application. Since the implementation of communications is architecture and system dependent, this file has to be provided for every (architecture, system) pair. As a start point we provide a simple implementation of push/pop operations that implement communications through shared memory, using statically allocated FIFO buffers. This simple implementation can be used on single-core architectures and multi-core architectures with shared memory.

4.2 Benchmark construction process

In order to extract the above information for each benchmark, we relied on the StreamIT compilation tools as much as possible and we then adapted their output to fit our needs. As presented by the dashed line in Figure 2, we modified the Java pretty-printer to generate a preliminary C version of the streaming application that later needs to be modified manually to match the analysis requirements. When finalizing the C source code through handmade modifications, we stayed independent from any specific run-time library and inter-core communication mechanism. Despite the error proneness of this method, this hand-made step is necessary to guaranty easy read/analyze/understand code with all required annotations. To validate the functional correctness of the final C source version, we performed non-regression tests considering the Java simulator output as the baseline.

Table 1 Description of provided benchmarks.

Name	#tasks	#split-join	Description
802.11a data rate 6/9/12/18/24/36	[119;132]	[17;18]	802.11a wireless LAN protocol transmitter with different configurations
Audiobeam	20	1	Real-time beam-forming on a microphone input array
Beamformer	56	2	Application to perform beam-forming on a set of inputs
CFAR	4	0	Constant False Alarm Rate detection
Complex-FIR	3	0	FIR filter with complex data types
DCT2	40	2	Discrete Cosine Transforms from Asplos'06 paper super-set
DES	423	80	DES encryption algorithm
FFT2	26	1	Fast Fourier Transform, blocked, coarse-grained version
FFT4	42	10	Fast Fourier Transform, more fine-grained
FilterBankNew	52	1	Creates a filter bank to perform multi-rate signal processing
FMRadio	43	7	FM radio with multi-band equalizer

To create the XML description, we needed the WCET of each task, the amount of data exchanged between task and the topology of the application's graph. For the first information, we relied on our tool Heptane [5] that gives us the WCET of each task in isolation. The amount of data exchanged and the topology of the graph are extracted manually from files generated by the Java simulator.

5 Provided benchmarks

Table 1 summarizes the benchmarks that are ready to use at the time of writing. The first column presents the name of the benchmark (identical to the name in the original StreamIT benchmark suite), followed by the number of tasks, the number of *split-join* nodes and a quick description (also extracted from the original StreamIT benchmark suite). For application 802.11a coming in multiple versions (to be explained later), we provide the minimum and maximum of provided values among all versions.

Table 2 shows the complexity of each benchmark. After the name of the benchmark, the second column shows the width of the graph (the maximum number of tasks at the same topological rank) which gives an idea of the amount of concurrency in the application. Following are information about task's WCET and amount of data exchanged between tasks. Both fields are described with an average and standard deviation.

Table 3 indicates which benchmarks need a *mathematic* library to compile, and use input and/or output file. Nonetheless to ensure self-containment, we provide a dummy implementation (empty shell) for the needed functions.

We found some benchmarks with multiple usages of the same task with different input parameters at different points in the application. We thus generated two versions of each benchmark: one with shared code to allow cache reuse, and another one with duplicated code. The difference between both versions lies on the ability to exploit cache reuse or not, and also accuracy of flow-facts annotations (which are more precise with duplicated code).

Table 2 Statistics for provided benchmarks.

Name	Width	WCET (cycles)	Data (tokens)	#Basic blocks (avg #instr.)	#Cond. br.	#mem. instr.
		<avg, standard deviation>				
802.11a 6/9/12/18/24/36	[7;18]	2.39e5, 6.35e5	596, 2238	1584 (6)	222	3519
Audiobeam	15	273, 1094	3, 5	386 (7)	72	893
Beamformer	12	1.25e4, 9.65e4	4.6, 10	459 (10)	87	1188
CFAR	1	1.58e4, 1.55e4	288, 425	375 (6)	70	821
Complex-FIR	1	501, 655	1.3, 0	336 (6)	60	804
DCT2	16	1.69e4, 3958	57.6, 91	437 (6)	86	894
DES	8	2045, 1417	35.7, 29	621 (6)	111	1079
FFT2	2	2.94e5, 3.03e5	137.8, 49	477 (6)	81	1003
FFT4	2	1337, 450	23.6, 8	566 (5)	85	860
FilterBankNew	6	6315, 8306	8.9, 11	446 (6)	84	913
FMRadio	12	1632, 2234	1.6, 1	486 (6)	91	1037

Table 3 Properties of provided benchmarks.

Name	Use math library	Use I/O file	Two versions / code reuse
802.11a	yes	no	yes
Audiobeam	yes	yes	no
Beamformer	yes	no	no
CFAR	yes	no	no
Complex-FIR	no	yes	no
DCT2	yes	yes	no
DES	no	no	yes
FFT2	yes	no	no
FFT4	no	no	no
FilterBankNew	no	no	no
FMRadio	yes	no	no

The last column of Table 3 indicates whether we created multiple versions of the benchmark with code reuse or not.

Finally, some benchmarks are customizable by modifying the value of some parameters inside the StreamIT source code, e.g. the data rate of the *802.11a* application. As modifying such values has an impact on the application's structure, we generated multiple versions of the same benchmark for the different configurations.

We successfully compiled the list of benchmarks presented in Table 1 for x86_64 architecture and validated their behavior by comparing their results with the one from the Java simulator provided by StreamIT. All these benchmarks are available at

`https://gitlab.inria.fr/brouxel/STR2RTS`.

6 Conclusion

This document has presented a collection of benchmarks written in analyzable C language and based on the StreamIT benchmarks suite [21]. The purpose of the refactoring of StreamIT

applications we have performed is to create self-contained analyzable architecture-independent parallel C applications to allow any kind of experiments on WCET analysis and real-time scheduling on multi-core architectures. To largely spread our work, we will integrate this collection of benchmarks into the TACLeBench project.

Due to the required handmade refactoring, we will be continuously adding new test cases over time while there are still some StreamIT applications to refactor. We foresee the end of refactoring in a couple of year for the 60% remaining benchmarks.

References

1 Matthias Becker, Dakshina Dasari, Borislav Nikolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *ECRTS*, 2016.

2 Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

3 Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258. IEEE, 1995.

4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

5 Hardy Damien, Rouxel Benjamin, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, 2017. doi:10.4230/OASIcs.WCET.2017.8.

6 Yorick De Bock, Sebastian Altmeyer, Jan Broeckhove, and Peter Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. In *Proceedings of the Embedded Operating Systems Workshop : EWiLi 2016*, pages 1–6. CEUR Workshop proceedings, October 2016.

7 Debie1. URL: https://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:debie1.

8 Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. URL: http://ziyang.eecs.umich.edu/~dickrp/e3s/.

9 Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

10 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET'16)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

11 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Bj"orn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Brussels, Belgium, July 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2010.136.

12 Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. Parmibench-an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, 9(2):45–48, 2010.

13 C. G. Lee. UTDSP Benchmark Suite, July 2011. URL: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

14 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006. doi:10.4230/OASIcs.WCET.2006.678.

15 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 309–318, 2014.

16 Parasuite. URL: http://parasuite.inria.fr/.

17 Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012. URL: http://www.cs.ucla.edu/pouchet/software/polybench.

18 Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

19 Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded java benchmark suite jembench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES'10, pages 120–127, New York, NY, USA, 2010. ACM. doi:10.1145/1850771.1850789.

20 S. Stuijk, M. C. W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. URL: http://www.es.ele.tue.nl/sdf3, doi:10.1109/ACSD.2006.23.

21 William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

Best Practice for Caching of Single-Path Code*

Martin Schoeberl[1], Bekim Cilku[2], Daniel Prokesch[3], and Peter Puschner[4]

1 **Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark**
masca@imm.dtu.dk
2 **Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria**
bekim@vmars.tuwien.ac.at
3 **Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria**
daniel@vmars.tuwien.ac.at
4 **Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria**
peter@vmars.tuwien.ac.at

Abstract

Single-path code has some unique properties that make it interesting to explore different caching and prefetching alternatives for the stream of instructions. In this paper, we explore different cache organizations and how they perform with single-path code.

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

Keywords and phrases single-path code, method cache, prefetching

Digital Object Identifier 10.4230/OASIcs.WCET.2017.2

1 Introduction

Worst-case execution time (WCET) analysis is a non-trivial analysis problem. It becomes especially difficult with more complex processor architectures. A strategy to simplify WCET analysis is to write programs that have a constant execution time, i.e., the best-case and worst-case execution time are equal. In that case, we do not need to analyze the program, but can simply measure the execution time. Single-path code gives constant execution time [15].

Single-path code is code that is structured so that there are no data dependent control flows. On an `if/else` condition both conditions are executed. However, to retain the program's semantics and data flow, all instructions are executed with a predicate. The compiler sets these predicates according to the original conditions of the branching code. When executing single-path code, instructions whose predicate evaluates to `false` do not update the processor state, i.e., they act as `nop` instructions. Loops always execute the maximum number of iterations (the so-called loop bound), which is a known number in a real-time context. Like the `if/else` case, the original loop condition is used to evaluate to a predicate and all instructions within loops are predicated.

* The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT (http://predict.compute.dtu.dk/), contract no. 4184-00127A. This paper was partially funded by the EU COST Action IC1202: Timing Analysis on Code Level (TACLe) and the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Single-path code can be manually coded or a compiler can translate *normal* code to single-path code. The translation of an `if/else` condition is also a common technique in compilers applied on small code fragments to avoid expensive branches. This is called if conversion [1].

The time-predictable execution of single-path code demands two features from a processor: (1) the processor needs to support predicates or a conditional move and (2) a predicated instruction shall have the same execution time irrespective of whether the predicate evaluates to `true` or `false`. Therefore, we explore single-path code on a processor that fulfills both conditions. Patmos is a processor that is designed especially for real-time systems [21]. Patmos contains features that shall make WCET analysis simpler, but also supports execution of single-path code with a predicated instruction set and constant execution time of instructions.

Patmos contains also a special instruction cache that caches full functions [4]. For historical reasons this cache is named method cache (it appeared first in a Java processor [18]). Cache misses can only occur at function calls or returns. Caching full functions has one drawback: code that is not executed is still loaded into the cache. However, as programs organized as single-path code execute all their instructions, this main drawback disappears. Therefore, our hypothesis is that the method cache is a good cache organization for single-path code.

This paper explores the method cache in the context of single-path code. We compare and evaluate the method cache against a standard instruction cache using the TACLeBench benchmarks [6]. Furthermore, we explore performance benefits of an extension of a standard instruction cache with a prefetcher that has been especially designed for single-path code.

The paper is organized in 6 sections: The following section presents related work. Section 3 provides background on single-path code generation and the time-predictable Patmos processor. Section 4 describes different options of caching for single-path code. Section 5 evaluates the different caching options on the Patmos processor and compares them. Section 6 concludes the paper.

2 Related Work

For real-time systems, caches are one of the main sources of temporal uncertainty. State-of-the-art cache analysis tools are using abstract interpretation for classifying cache accesses [12]. However, even if these approaches derive safe bounds, the precision of the results derived from the abstract models strongly vary depending on the cache architecture and replacement policy [9]. For example, an abstract model for the LRU replacement policy achieves better predictability than a model for FIFO or PLRU [17].

Another mechanism that aims at making caches more predictable is cache locking [14]. This technique loads memory contents into the cache and locks it to ensure that it will remain unchanged afterwards. The benefit of cache locking is that all accesses to the locked cache lines will always result into cache hits. The cache content can be locked entirely [7] or partially, it can be locked for the whole system lifetime (static locking) or it can be changed at runtime (dynamic locking) [5]. Although cache locking increases predictability, it reduces performance by restricting the temporal locality of the cache to a set of locked cache lines.

In contrast to conventional code, single-path conversion overcomes predictability issues by generating code that has only a single trace of execution. Thus, keeping traces of possible cache states is no more needed. Furthermore, the use of single-path code eliminates the necessity for cache locking.

3 Background

This paper builds on prior research work on single-path code and research on the time-predictable computer architecture developed for the T-CREST platform [19].

3.1 Single-path Code Generation

Puschner and Burns propose single-path code to simplify WCET analysis by avoiding data-dependent control flow decisions [15]. The defining property of single-path code is that any execution follows a single instruction trace, independent from input data. This is achieved by conversion of control dependence to data dependence, with the use of predicated instructions. In code that is WCET analyzable, loops must be bounded. The compiler transforms input-data dependent loops such that they iterate for a fixed number of times, which is the local loop bound [13].

The resulting single-path code may have a longer execution path, due to the serialization of otherwise alternative paths in the original program. Also, in some scenarios it is undesirable to always consume the computational resources required for the worst-case, e.g., in mixed-critical systems where slack time is used for non-critical tasks. Nonetheless, single-path code generation provides a constructive approach to time-predictable real-time code. On a "well-behaved" hardware platform, the execution time for single-path tasks is constant. In this ideal case, WCET analysis simplifies to measurement.

One requirement is that the instruction timing is independent of the instruction operands. Memory accesses introduce another source of variability in execution time. Though, the single-path property makes the code easier to analyze with regards to instruction memory. Abstract interpretation based analysis becomes superfluous, there is no need for approximation. The known singleton instruction stream can be directly applied to a hardware model of the instruction cache (as in simulation). This knowledge is exploited to implement perfectly accurate prefetching schemes for instructions [2].

Data accesses are also subject to execution-time variability. Enforcing local availability of the required data during the task execution may alleviate the problem, e.g., by data cache locking or usage of a scratchpad memory. However, we restrict ourselves to the instruction cache in this paper.

3.2 Patmos and the T-CREST Platform

We explore instruction caching options on the Patmos processor [21], which itself is part of the T-CREST multicore platform [19]. The T-CREST platform aims to build a processor, network-on-chip, and compiler toolchain [16] to simplify WCET analysis. We optimized all components to be time-predictable, even when average-case performance is reduced. AbsInt aiT [8] static WCET analyzer supports the Patmos processor. T-CREST also includes the research WCET analyzer platin [11].

Patmos is a RISC architecture supporting dual-issue instructions. To the best of our knowledge, Patmos is timing anomaly free. There is no timing dependency between any two instructions. Even all cache misses (instruction or data) happen in the same pipeline stage (the memory stage). Therefore, only a single cache miss can happen any clock cycle. Patmos uses special forms of instruction and data cache that shall simplify cache analysis. For instructions, Patmos has a method cache [4], which caches whole functions. Besides these special caches Patmos also supports a standard instruction cache, a standard data cache, and instruction and data scratchpad memories.

One issue with a method cache is that complete functions are loaded into the method cache, even when only part of it is executed. We attack this issue by splitting larger functions into smaller subfunctions [10]. However, with single-path code there is no code that is not executed. The processor executes all instructions of a called function. Therefore, a method cache may well fit for caching single-path code, especially when callee functions are not evicted by called functions.

We extended a standard instruction cache by a prefetching unit [3] to improve single-path execution time. The prefetcher uses the static "knowledge" of single-path code to anticipate the upcoming instructions and bring them into the cache before they are needed. Such a property allows the prefetcher to perform in time-predictable fashion without polluting the cache with unused instructions.

4 Caching of Single-Path Code

Single-path code is instruction-cache friendly as all instructions that are loaded into the cache are executed, except at the end of a function.

4.1 Standard Instruction Cache

A standard instruction cache is organized in cache blocks and can be configured as direct mapped cache or set associative cache. Some single-path code can benefit from a direct mapped instruction cache, especially when the cache is small [3]. We will evaluate direct mapped and set-associative instruction caches for single-path code.

4.2 Method Cache

The method cache is an instruction cache designed to simplify WCET analysis. The method cache caches full functions/methods. Therefore, a cache miss can only happen on a call or a return. All other instructions are guaranteed hits and cache analysis can ignore those. Method cache analysis only needs to consider functions and not individual instructions.

One disadvantage of the method cache is that instructions in a function that are not executed are still loaded on a cache miss. However, with single-path code all instructions of a function are always executed. Therefore, the method cache should perform well with single-path code.

4.3 Time-predictable Prefetcher with a Standard Cache

Prefetching hides large memory access latencies by loading instructions into the cache before they are needed. However, to take advantage of this improvement, the prefetcher needs to guess the right prefetch target and issue the request at the right time. Any wrong speculation on these two parameters can degrade the system performance.

The time-predictable prefetcher exploits properties of single-path code to anticipate the future instruction cache accesses with full accuracy and bring those instructions into the cache at the right moment [3]. For higher efficiency, the prefetcher implements an aggressive algorithm that prefetches every cache line of the code. Its direction is controlled through a *Reference Prediction Table* (RPT), which is an optimized projection of single-path code that captures the control-flow behavior of the code.

The entries of the RPT control the behavior of the prefetcher. They contain addresses at which the prefetcher should switch between sequential and non-sequential prefetching.

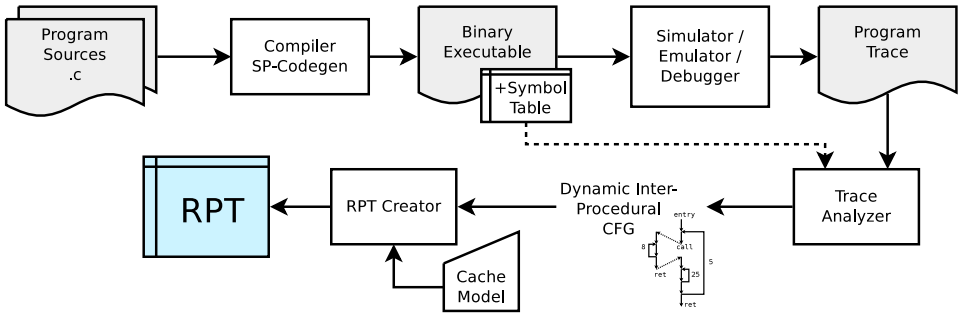


Figure 1 Generation of the Reference Prediction Table (RPT).

Figure 1 shows the generation of the RPT. It begins with obtaining the execution trace of the single-path code. We use the Patmos simulator to export the program counter values during a program run. We extract the start addresses of the functions from the symbol table of the executable. The trace analyzer uses the trace and the start addresses to produce a dynamic control-flow graph of the single-path function, where nodes are addresses of single instructions. The trace analyzer identifies call sites, loops, loop nests, and loop iteration counts. The RPT creator then creates entries containing an address that shall trigger a change in the behavior of the prefetcher, a destination where to continue prefetching, and additional information depending on the entry type.

Generation of the RPT is based on a form of dynamic program analysis, without instrumentation of the original program. However, the extracted instruction trace is invariable regardless of the program inputs because of the single-path code. Also, in contrast to trace-based and profile-guided optimization, the analysis results are not directed back to the compiler. Instead, the execution is optimized by means of hardware in form of the specialized prefetcher.

5 Evaluation

We evaluate the program performance of single-path code with different caching methods. For the comparison, we use the Patmos processor. We configure Patmos for the Altera DE2-115 FPGA board, which means that the main memory is a 16-bit SRAM. This memory results in 21 clock cycles for a burst of 4 32-bit words to fill or spill a 16-byte cache line. All standard caches have the line size of the burst length, 16 bytes. We configure the instruction or method cache to be 8 KB large and the method cache to cache up to 16 functions. The data cache is 4 KB and the stack cache 2 KB. We use hardware simulation to get cycle accurate measurements.

For the evaluation, we use the TACLeBench benchmark collection [6] in version 1.9. We have added an attribute to the benchmarks' main function to avoid that it is inlined by the compiler. Otherwise we did not touch the source of TACLeBench. This main function is also the root function for the single-path code generation. We measure the execution time of the whole program, including initialization and result comparison code, in clock cycles.

We used a subset of the benchmarks. The variation of the execution time of the benchmarks is high, i.e., between hundreds and a billion clock cycles. For practical reasons, we did not use the long running benchmarks, as cycle accurate hardware simulation is time consuming.[1]

[1] The simulation of the remaining benchmarks, just for a single cache size configuration, still takes 6–8 hours on a contemporary notebook.

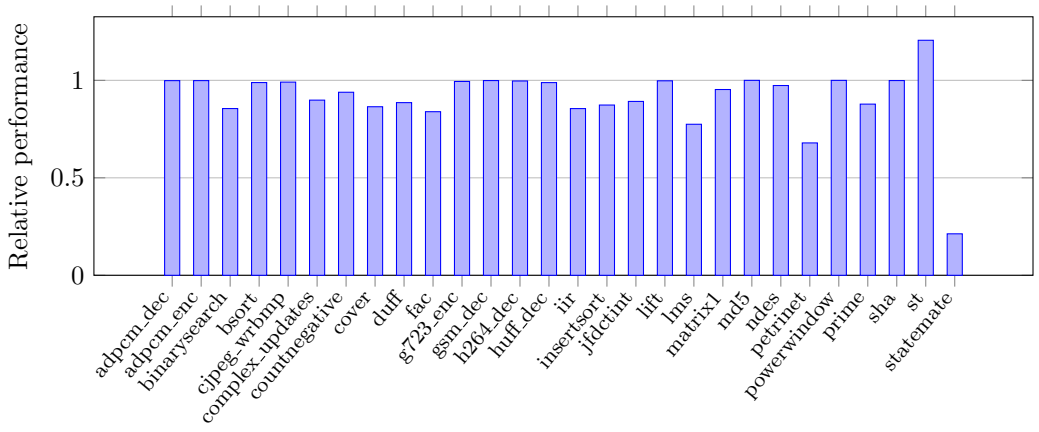


Figure 2 Relative average-case performance comparing the method cache with a standard cache on normal programs.

However, a large execution time does not necessarily mean that those benchmarks would have a larger memory foot print. Furthermore, we dropped benchmarks where we cannot generate single-path code, e.g., recursive benchmarks. Also, we removed two outliers (`ludcmp` and `minver`) as their results showed improvements of factors 3 to 4 for the method cache compared to a standard instruction cache.

All the following figures show measured performance improvements, where a bigger number means a better result. The figures show relative performance as one execution time scaled by the other execution time. That means a number larger than 1 is an improvement, and a number less than 1 is a regression. E.g., when comparing the method cache with the standard cache, the figure shows how much better (or worse) the configuration of the method cache is compared with the standard cache. For this example, the relation is the execution time of the benchmark with the standard cache divided by the execution time of the benchmark with the method cache (as a shorter execution time is better).

5.1 Baseline

As a baseline, we show the performance difference between using a method cache and a direct mapped instruction cache on normal compiled code. Figure 2 shows the execution time relation between those two configurations (normalized to the execution time with the standard cache). The geometric mean of the comparison is 0.88, which means that on average the configuration with the direct mapped instruction cache performs better. Those measurements are average case measurements and cannot be an indication of WCET analysis bounds. In these average case measurements, we see that some benchmarks perform equally for the two cache configurations. We assume those cases are when the benchmark fits entirely into the cache. Several benchmarks perform better with a normal instruction cache than with the method cache. However, this is an average case measurement and the method cache was designed to simplify WCET analysis.

5.2 Single-Path Comparison and Prefetching

Figure 3 shows the performance comparison between a method cache and a standard cache with single-path generated code. The figure is now more diverse than the average-case figure.

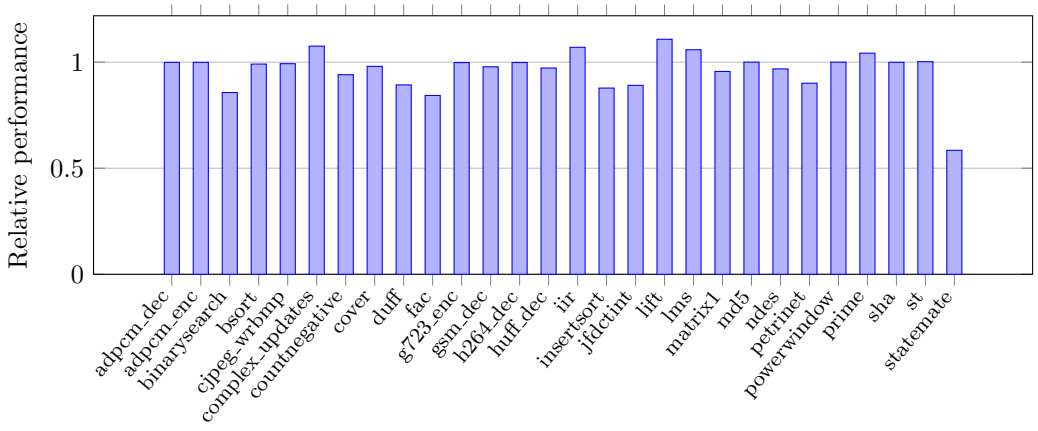


Figure 3 Relative single-path performance comparing the method cache with a standard cache.

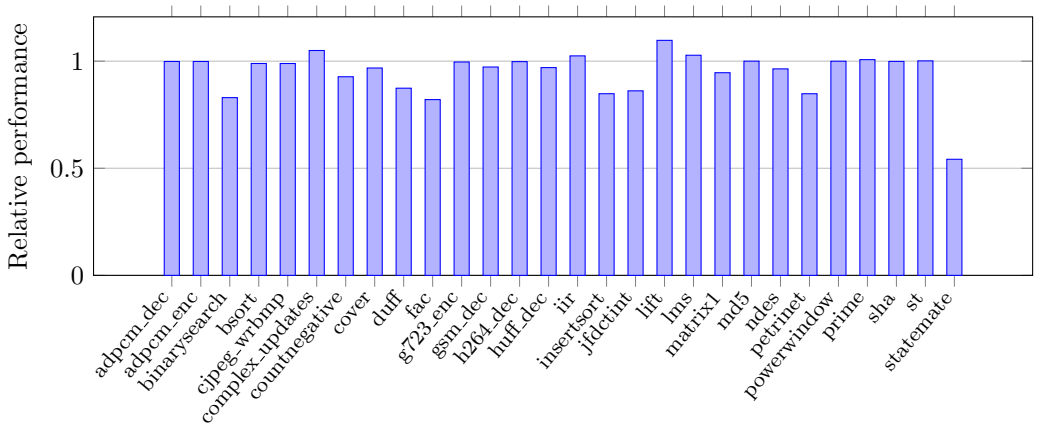


Figure 4 Relative single-path performance comparing the method cache with a prefetching cache.

Some benchmarks gain and some lose when using a method cache. There is no clear winner. The geometric mean of the comparison is 0.96, which means that on average the standard cache configuration performs slightly better.

Figure 4 show the performance comparison between a method cache and an instruction cache that includes the prefetching unit. The results are similar to the results in Figure 3. Some benchmarks gain a little bit with the prefetching unit. The geometric mean of the comparison is 0.94. We assume that most benchmarks are almost fitting into the cache and leaving not enough room for improvement by prefetching. It has been shown that smaller caches benefit most from the prefetcher [3].

5.3 Associativity

Figure 5 shows the comparison of a 2-way cache with LRU replacement with a direct mapped instruction cache. Originally we assumed that a direct mapped cache is a better fit for single-path code as is avoids cache thrashing on loops that are larger than the cache. However, we see in the figure that some benchmarks benefit from a higher associativity. Only `statemate` performs better with a direct mapped cache. Therefore, we deduct that the 4 KB of one way is large enough for the larger loops in the benchmarks. The geometric mean of the comparison

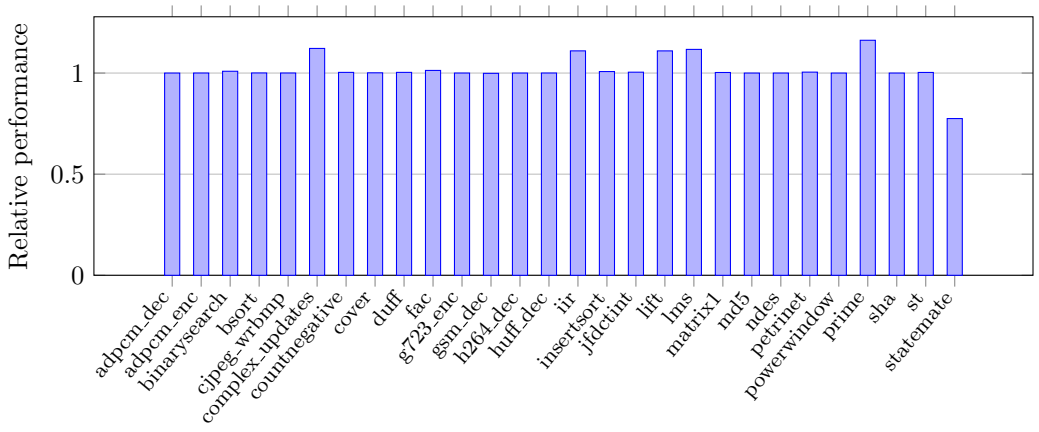


Figure 5 Relative single-path performance comparing a 2-way cache with a direct mapped cache.

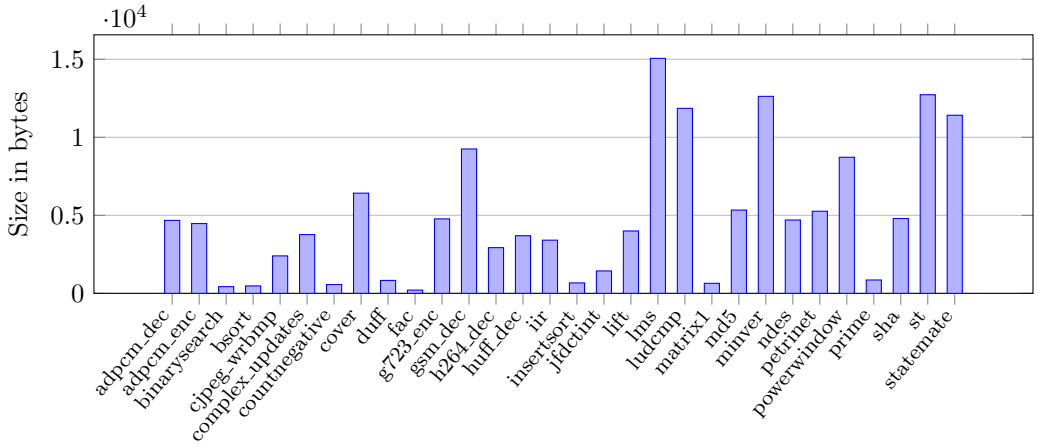


Figure 6 Benchmark dynamic sizes (instruction memory footprint).

is 1.014, which means that on average the configuration with the 2-way instruction cache performs slightly better than the configuration with the direct mapped cache.

5.4 Benchmark Sizes

To better understand the benchmark results, we measure the instruction memory footprint of the benchmarks. We collect actual runtime data of which code is executed. We trace the execution of a benchmark with the Patmos simulator and collect which functions are executed. We extract the function sizes from the executable. As single-path code executes the whole function, we simply add all sizes of the executed functions to retrieve the instruction memory footprint. We exclude startup and exit code.

Figure 6 shows the memory footprint of the benchmarks. We can see three typical sizes of benchmarks: (1) very small benchmarks, such as `binarysearch` or `duff` where the memory footprint is less than 1 KB, (2) medium sized benchmarks with a memory footprint of 4–5 KB, and (3) larger benchmarks with a memory footprint of around 10 KB. The largest benchmark is `lms` with 14.7 KB. These numbers do not include any startup or exit code executed, which by itself is 7048 bytes. That means for the small benchmarks the startup and exit code dominates the memory footprint.

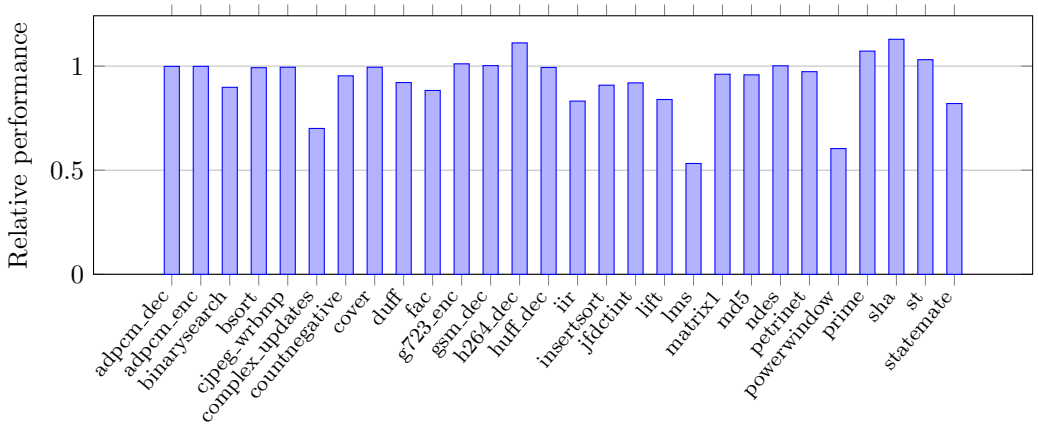


Figure 7 Relative single-path performance comparing the method cache with a standard cache (2 KB size).

With our standard configuration of 8 KB instruction cache, 23 out of the 30 benchmarks will fit into the cache (excluding startup and exit code). This means that many of the TACLeBench benchmarks are too small for the evaluation of different instruction caches. Therefore, we explore artificial small caches in the following subsection.

5.5 Small Caches

For further experiments we reduce the size of the instruction and method cache to just 2 KB, a very small size for current embedded processors. Figure 7 compares the method cache with the direct mapped instruction cache. For this evaluation we leave the compiler parameter preferred subfunction size at the default value of 256. The geometric mean of the comparison is 0.92, which means that on average a direct mapped instruction cache is a slightly better solution for small caches.

Figure 8 compares a 2-way cache with LRU replacement with a direct mapped instruction cache. Compared to Figure 5 we see different benchmarks benefitting from associativity. Furthermore, with the larger cache only a single benchmark (`statemate`) performed worse with the 2-way cache. For the small configuration, several benchmarks perform worse with a 2-way set associative cache. Therefore, for small caches there is no clear winner when comparing a direct mapped and a 2-way set associative instruction cache. The geometric mean of the comparison is 1.05, which means that on average a 2-way set associative instruction cache is a slightly better solution than a direct mapped instruction cache for small cache sizes.

5.6 Discussion

Single-path code has different characteristics than normal code. We see some of the different characteristics when comparing different caching methods. The method cache, which works not so well in the average-case, is a better fit when using single-path code. Prefetching with a standard cache provides some benefit.

As we see in the results, there is no clear winner for all benchmarks. Therefore, if we use an FPGA as execution platform, we can select an application specific caching method. This is like an application specific instruction set in a processor.

It might also be interesting to compare the different caching methods for single-path code with either cache locking or allocation in a scratchpad memory (SPM). In earlier work, we

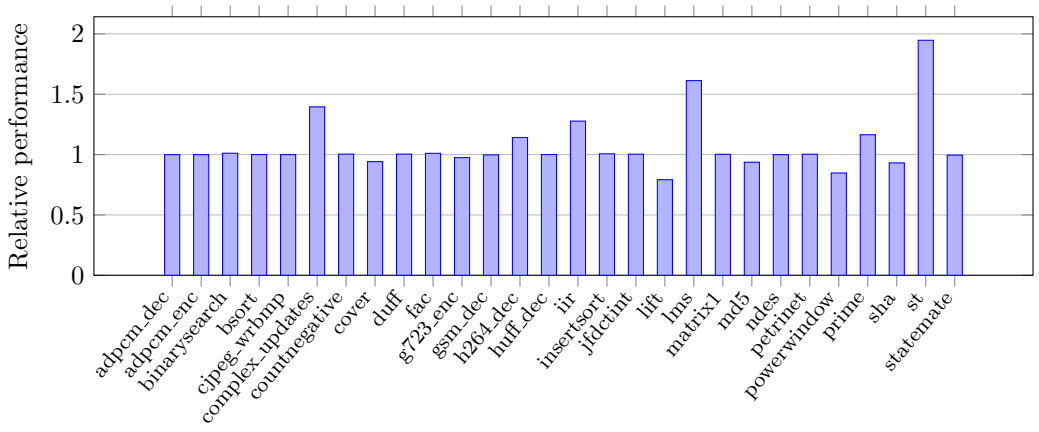


Figure 8 Relative single-path performance comparing a 2-way cache with a direct mapped cache (2 KB size).

compared two alternatives to a standard instruction cache: an SPM and a method cache [22]. The comparison considers the true WCET and the estimated WCET bound. We found that a method cache is preferable to an SPM for the true WCET. However, if WCET bounds are derived by analysis, the WCET bounds for an instruction SPM are often lower than the bounds for a method cache. As there is no WCET overestimation in single-path code, it would be interesting to repeat this comparison with single-path code.

5.7 Reproducing the Results

We think reproducibility is of primary importance in science. As we are working in the context of an open-source project, it is relative easy to provide pointers and a description how to reproduce the presented results.

The T-CREST project is open-source and the README[2] of the Patmos repository provides a brief introduction how to setup an Ubuntu installation for T-CREST and how to build T-CREST from the source. More detailed installation instructions, including setup on Mac OS X, are available in the Patmos handbook [20]. To simplify the evaluation, we also provide a VM[3] where all needed packages and tools are already preinstalled. However, that VM is currently used in teaching and does not contain the latest version of T-CREST, including the scripts for the experiments. Therefore, you need to reinstall and build T-CREST as described in the README.

We have scripted all experiments and host those scripts in the `misc` repository of T-CREST. Details to rerun the experiments are described in a README.[4] The `Makefile` is setup to run the base experiments and for producing the figures as PDFs. Variations can be obtained by changing the respective variables.

[2] `https://github.com/t-crest/patmos`
[3] `http://patmos.compute.dtu.dk/`
[4] `https://github.com/t-crest/patmos-misc/tree/master/experiments/cache_prefetch_lock/wcet2017`

6 Conclusion

In this paper, we compared different caching methods for single-path code. We found that the method cache, which performs not so well in the average case, shows an improvement on some benchmarks when compared to a standard instruction cache. With small cache configurations the best solution is in most cases a set associative instruction cache. When we use an FPGA as execution platform we have the freedom to choose the best caching solution for each individual application.

References

1 J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

2 Bekim Cilku, Daniel Prokesch, and Peter Puschner. A time-predictable instruction-cache architecture that uses prefetching and cache locking. In *Proc. 18th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) Workshops, 11th IEEE/IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 74–79. IEEE CS Press, 2015.

3 Bekim Cilku, Wolfgang Puffitsch, Daniel Prokesch, Martin Schoeberl, and Peter Puschner. Improving performance of single-path code through a time-predictable memory hierarchy. In *Proceedings of the 20th IEEE International Symposium on Real-Time Computing (ISORC 2017)*, Toronto, Canada, May 2017. IEEE.

4 Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A Method Cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE. doi:10.1109/ISORC.2014.47.

5 Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

6 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)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

7 Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148. ACM, 2007.

8 Reinhold Heckmann and Christian Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. [Online, last accessed November 2013]. URL: http://www.absint.de/aiT_WCET.pdf.

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

10 Stefan Hepp and Florian Brandner. Splitting functions into single-entry regions. In Karam S. Chatha, Rolf Ernst, Anand Raghunathan, and Ravishankar Iyer, editors, *2014*

International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2014, Uttar Pradesh, India, October 12-17, 2014, pages 17:1–17:10. ACM, 2014. doi:10.1145/2656106.2656128.

11 Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

12 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

13 Daniel Prokesch, Benedikt Huber, and Peter P. Puschner. Towards automated generation of time-predictable code. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASICS*, pages 103–112. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

14 Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 114–123. IEEE, 2002.

15 Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society. doi:10.1109/WORDS.2002.1000040.

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

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

18 Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer. doi:10.1007/b102133.

19 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil 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. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

20 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

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

22 Jack Whitham and Martin Schoeberl. WCET-based comparison of an instruction scratchpad and a method cache. In *Proceedings of the 10th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2014)*, Reno, Nevada, USA, June 2014. doi:10.1109/ISORC.2014.48.

On the Representativity of Execution Time Measurements: Studying Dependence and Multi-Mode Tasks

Fabrice Guet[1], Luca Santinelli[2], and Jerome Morio[3]

1 **ONERA Toulouse, Toulouse, France**
2 **ONERA Toulouse, Toulouse, France**
3 **ONERA Toulouse, Toulouse, France**

Abstract

The Measurement-Based Probabilistic Timing Analysis (MBPTA) infers probabilistic Worst-Case Execution Time (pWCET) estimates from measurements of tasks execution times; the Extreme Value Theory (EVT) is the statistical tool that MBPTA applies for inferring worst-cases from observations/measurements of the actual task behavior. MBPTA and EVT capability of estimating safe/pessimistic pWCET rely on the quality of the measurements; in particular, execution time measurements have to be representative of the actual system execution conditions and have to cover multiple possible execution conditions. In this work, we investigate statistical dependences between execution time measurements and tasks with multiple runtime operational modes. In the first case, we outline the effects of dependences on the EVT applicability as well as on the quality of the pWCET estimates. In the second case, we propose the best approaches to account for the different task execution modes and guaranteeing safe pWCET estimates that cover them all. The solutions proposed are validated with test cases.

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

Keywords and phrases Measurement-Based Probabilistic Timing Analysis, probabilistic Worst-Case Execution Time, Extreme Value Theory, Execution Time Measurements Representativity

Digital Object Identifier 10.4230/OASIcs.WCET.2017.3

1 Introduction

Multi-core and many-core processors are becoming common implementations for real-time systems. The large amount of available resources allows increasing performance and embedding multiple functionalities within systems. However, real-time modeling and analysis become more complex due to the increased source of unpredictabilities, [11, 22]; for instance, tasks execution time exhibits variabilities from the runtime dependence/interference between system elements which are difficult to model accurately e.g., access to shared mameory, [21].

Probabilistic timing analysis approaches are being proposed to cope with real-time system unpredictabilities. They consider both the task average execution behavior and the worst-case execution behavior as random variables. In particular, the probabilistic Worst-Case Execution Time (pWCET) extends the notion of Worst-Case Execution Time (WCET) as the worst-case distribution that upper bounds the task execution times. pWCET models have multiple values, each with an associated probability of being the task worst-case execution time; very unlikely cases such as faults are included. This makes pWCET task models more flexible and potentially less pessimistic than classical deterministic WCET (either statically or measurement-based driven) in representing the task behavior. Figure 1 gives an example

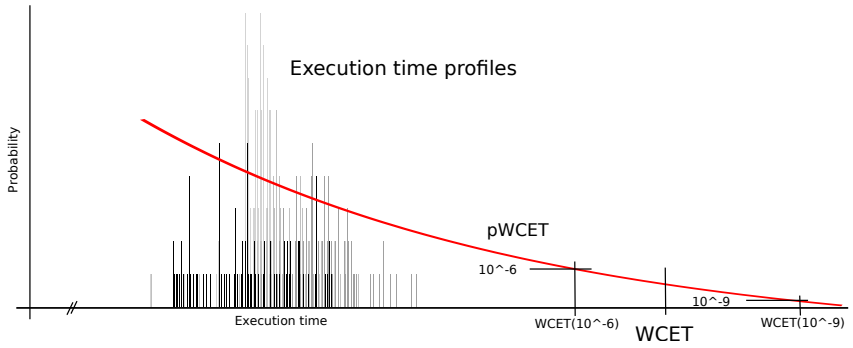


Figure 1 WCET and pWCET representations.

of pWCET and WCET representations to the task behavior; both upper bound the task actual execution times which could be different execution time profiles depending on the system execution conditions.

Measurement-Based Probabilistic Timing Analysis. The Measurement-Based Probabilistic Timing Analysis (MBPTA) is a probabilistic timing analysis that makes use of measurements of task execution times for computing pWCET estimates. The Extreme Value Theory (EVT) applied in the MBPTA allows for inferring the rare events (worst-case bounds) from observations of the actual task behavior (measurements). MBPTA does not need accurate system nor task models, instead they demand measurements of execution time representative of all the system execution behaviors.

A first application of the EVT for the timing analysis of real-time systems considers Gumbel distributions for the pWCET estimates, [4]. In [6, 1, 5], only artificially[1] time randomized real-time systems are analyzed with the EVT. Last developments in MBPTA propose a generalized version of the EVT [18, 12] which can be applied to both non-time randomized real-time systems and artificially time randomized real-time systems.

MBPTA Open Problems. Today's MBPTA works have completely defined the EVT and its applicability to the pWCET problem. The hypotheses for applying the EVT have been deeply investigated, and the quality (as safety and accuracy) of the resulting pWCET estimates has reached good levels. Actual MBPTA challenges are moving to the *representativity* of the execution time measurements, since in order to let the EVT be able to estimate safe worst-case distributions[2], the measurements have to be "good representation" of the system behavior,[20]. We hereby consider a notion of measurement representativity as the capability of capturing any event that characterizes the current system behavior. Those events would be dependence between consecutive executions, pattern of executions e.g., cyclic execution times or clusters, multiple execution conditions or operational modes, etc..

In [1], the system is artificially randomized in order to make the appearance of worst-case measurements more probable. For those systems, some works have approached the problem of measurement representativity [2, 20] where the representativity notion considered restricted

[1] By artificially time randomized systems we mean systems where there have been added randomization mechanisms such as random replacement caches or random task re-mapping in memory at each execution.
[2] Safe worst-case execution time distributions (pWCETs) are distributions that upper bound any possible task execution.

to the capability of measurements of capturing worst-case events. instead, we hereby consider a broader notion of representativity that includes the capability of capturing dependence or pattern of executions.

Full coverage of tasks/system input conditions and measurements, which needs to include pathological cases and their large execution times, have to be guaranteed to MBPTA. As today, they remain open problems related to the representativity of the measurements of any architecture, including artificial randomized real-time systems.

In this paper we focus on two aspects of the representativity which real-time systems face constantly: statistical dependence between measurements and tasks with multiple operational modes.

We intend to demonstrate that measurements which are representative of the dependent system behavior have to be preserved and not modified whatsoever. Instead, with measurements which are representative of multiple execution condition, worst-case behaviors or modes cannot be neglected.

- The statistical dependence between measurements is when from one set of measurements it is possible to infer future measurements e.g., clusters of measurements or consecutive measurements with similar values appearing periodically. With real-time systems, examples of dependences are series of specific execution conditions e.g., bursts of interferences or cache locality, or task inputs/execution conditions that appear periodically. *Would it possible to apply the EVT in case of dependence between measurements? What is the impact that dependences have on pWCET estimates?*
- Real-time tasks can be implemented with multiple operational modes e.g., taking-off, cruising and landing modes which alternate at runtime in avionic systems. More simple examples are multi-path tasks where, depending on the input applied a path can be triggered with consequently different execution time. *How is it possible to apply the EVT to multi-mode execution time measurements? What is the pWCET estimate that guarantees all the modes, a.k.a. a safe pWCET estimates?*

If the system shows dependent behavior and multi-mode tasks, the measurements have to embed such events in order to characterize the system behavior and being representative for it. To the previous questions we provide answers with this work.

Contributions: We propose guidelines for letting EVT and MBPTA tackle with dependent measurements of execution times and multi-mode real-time tasks. We describe what has to be done in both cases in order to correctly apply the EVT and obtain safe and accurate pWCET estimates. We provide also a statistical analysis to the measurements for identifying the limits of the EVT application and the conditions for qualitatively defining the representativity of the measurements in terms of dependence and multi-mode tasks. This would allow to extend EVT applicability to more realistic real-time systems e.g., with statistical dependence or multi-path tasks. Our contributions are validated with test cases from industrial applications, multi- and many-core real-time systems and artificial traces of execution time measurements.

Organization of the paper: In Section 2 we state some background for the MBPTA, the EVT and the probabilistic modeling of average and worst-case task execution behavior. Section 3 details the EVT applicability in case of statistical dependent measurements and the effects that dependence has on pWCET estimates; case studies are applied to validate the guarantees that EVT offers to task pWCETs in case of dependence. Section 4 approaches the challenge of EVT applicability to multi-mode tasks; solutions to guarantee pWCET

estimates with multiple execution conditions are developed and validated with case studies. Section 5 is for conclusions and future work.

2 Background: Probabilistic Modeling and Extreme Value Theory

A trace $\mathcal{T}$ is a collection of execution time measurements C_j, $\mathcal{T} = \{C_j \mid j \in [\![1 : n]\!]\}$, n is the size of the trace. Given $\mathcal{T}$, the Execution Time Profile (ETP) $\mathcal{C}$ is the discrete random variable defined on the finite support $\Omega_\mathcal{C}$ of possible execution time values $C_{(k)}$, $\Omega_\mathcal{C} = (C_{(k)})_{k\in[\![1:N]\!]}$ with $C_{(k)} \in \mathcal{T}$; N is the number of different value in $\mathcal{C}$. The ETP is an empirical discrete random variable[3] that describes the task actual execution behavior. Representations for $\mathcal{C}$ are: the discrete probability mass function or Probability Distribution Function (PDF) $\mathsf{pdf}_\mathcal{C}$, such that $\mathsf{pdf}_\mathcal{C}(C_{(k)}) = P(\mathcal{C} = C_{(k)})$, the empirical Cumulative Distribution Function (CDF) $\mathsf{cdf}_\mathcal{C}$ as the discrete function $\mathsf{cdf}_\mathcal{C}(C) = P(\mathcal{C} \leq C)$ with $\mathsf{cdf}_\mathcal{C}(C) \in [0; 1]$ and the Complementary Cumulative Distribution Function (CCDF) $\mathsf{icdf}_\mathcal{C}$ defined by the probability of exceeding the execution time threshold C (risk probability), $\mathsf{icdf}_\mathcal{C}(C) = P(\mathcal{C} > C) = 1 - \mathsf{cdf}_\mathcal{C}(C)$ with $\mathsf{icdf}_\mathcal{C}(C) \in [0; 1]$.

Probabilistic timing analysis approaches look for pWCET distribution estimates $\overline{\mathcal{C}}$ that upper bounds any possible task execution behavior. Representations for $\overline{\mathcal{C}}$ are the PDF $\mathsf{pdf}_{\overline{\mathcal{C}}}(C)$ either continuous or discrete, the CDF $\mathsf{cdf}_{\overline{\mathcal{C}}}(C)$ and the CCDF $\mathsf{icdf}_{\overline{\mathcal{C}}}(C)$. $\overline{\mathcal{C}}$ has to be a *safe/pessimistic* representation of the task worst-case behavior: it has to be larger than or equal to[4] any ETP $\mathcal{C}^j$ the task can have, $\mathsf{icdf}_{\overline{\mathcal{C}}}(c) \geq \mathsf{icdf}_{\mathcal{C}^j}(c)$ for every c and every execution condition j. $\overline{\mathcal{C}}$ has also to be a *tight* upper bound to the ETPs; the tightness is for the quality of the pWCET estimates.

2.1 The Extreme Value Theory in a Nutshell

The EVT applies with the Block Maxima (BM) paradigm or with the Peak over Threshold (PoT) paradigm. The BM EVT models the limit law of execution time maxima of blocks of execution time measurements; the PoT EVT models the limit law of the execution times greater than a threshold (peaks above the threshold).

At the infinite (infinite number of block maxima or infinite number of peaks over the threshold) the law of extreme measurements tends to a Generalized Extreme Distribution (GED) or a Generalized Pareto Distribution (GPD) if and only if: i) the input $\mathcal{T}_\mathcal{C}$ is composed of independent and identically distributed (iid) measurements and ii) the resulting distribution $\mathcal{C}$ belongs to the Maximum Domain of Attraction (MDA) of the limit distribution. GED and GPD are the limit distribution respectively for the BM and the PoT EVT.

The identical distribution hypothesis assumes that all the measurements C_j follow the same distribution $\mathcal{C}$. The independence hypothesis (statistical independence) assumes that the individual execution time measurements $C_1, \ldots, C_n$ are not correlated with each other. The MDA hypothesis (also named matching) seeks if the limit law of the input distribution $\mathcal{C}$ converges to a GEV or a GPD. The limit law from the EVT is the pWCET estimates $\overline{\mathcal{C}}$.

Generalized EVT. Recent works prove the EVT applicability with more relaxed hypotheses than iid and MDA, [17, 12, 18]. They formalize the so called *generalized EVT* or practical EVT,

[3] $\mathcal{C}$ is a discrete distribution since execution time C_j can only assume values multiple of the system tick. Calligraphic letters are for both random variables, discrete or continuous, and traces, $\mathcal{C}$ and $\mathcal{T}$. Non-calligraphic letters are for single value variables C_j and $C_{(k)}$.

[4] The partial ordering between distribution is defined according to [8].

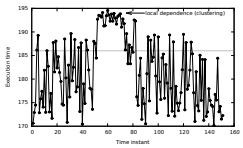


Figure 2 Local dependence between execution time measurements.

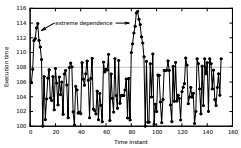


Figure 3 Extreme dependence between execution time measurements.

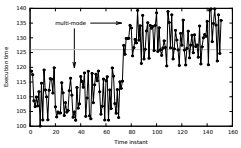


Figure 4 Example of two-mode task.

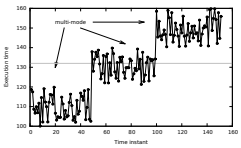


Figure 5 Example of three-mode task.

since it applies to practical cases of real-time systems e.g., not infinite measurements, system without artificially time randomization. The generalized EVT relies on: the stationarity hypothesis h'_1, the short range independence (negation of short range dependence) hypothesis $h'_{2.1}$, the extremal independence (long range dependence) hypothesis $h'_{2.2}$ and the matching hypothesis h'_3. If the EVT follows all the hypothesis, then it provides a safe estimation of the extreme execution times of $\mathcal{C}$.

The *stationarity hypothesis* h'_1 tests if the measurements are stationary and follow the same distribution i.e. the identical distribution. The Kwiatowski Philips Schmidt Shin (KPSS) test [13] checks if the trace is stationarity.

The dependence between measurements instantiates into local dependence i.e. close dependent measurements in $\mathcal{T}_\mathcal{C}$, and dependence between extreme measurements i.e. far measurements.

The *short range dependence (or local dependence)* $h'_{2.1}$ focuses on the relationship between measurements close-within-$\mathcal{T}_\mathcal{C}$. Condition D in [16] formalizes the minimum degree of short range dependence for the EVT applicability; it ensures that for distant enough dependent measurements (short range dependence), the limit law of the peaks over a threshold is still a GPD. A valuable test for the short range dependence $h'_{2.1}$ is the Brock Dechert Scheinkman (BDS) test [3]. Figure 2 gives an example of local dependence between measurements of execution times: execution times above a threshold can cluster as a result of dependence and consecutive interferences between system elements.

The *long range dependence (or extremal dependence)* $h'_{2.2}$ focuses on the relationship between far-in-time measurements. Condition D' in [16] formalizes the minimum degree of long range dependence for the EVT applicability. The extremal index θ, $\theta \in]0;1]$ [9], indicates the degree of clustering of either the PoT or the BM. θ expresses the probability of having distant enough measurements which are independent: the more the peaks or the maxima are distant from each other the more the independence is, and the higher is the probability of having independence it is. θ, with one of its estimators is applied to verify the extremal independence $h'_{2.2}$ [12, 10]. Figure 3 gives an example of extremal dependence between measurements of execution times; patterns that repeat are impacted one another.

The *matching hypothesis* h'_3 is for verifying that $\mathcal{C}$ belongs to the MDA of the GPD. A good matching test is the Cramer Von Mises criterion (CVM) which measures the distance between the empirical CDF of the extreme measurements and the $\overline{\mathcal{C}}$ estimated. The CVM test verifies the validity of h'_3 and it has been chosen because it performs well in the case of extreme value distributions [14].

DIAGXTRM [12] is a MBPTA tool that implements the generalized EVT and we use it to investigate statistical dependence and multi-mode tasks in this work. It implements the PoT EVT version as well as the tests previously described. It also defines confidence levels cl_i to verify the confidence on the EVT applicability hypotheses, thus the confidence

on the pWCET estimates; cl_i defines the confidence level on $h'_i \in H'$. cl_i is an integer value, $cl_i \in \mathcal{N}$, and defined in $[0, 4]$, $cl_i \in [0, 4]$, such that for $cl_i = 0$ there is no confidence in accepting h'_i; for $cl_i = 1$ there is moderate confidence in accepting h'_i; for $cl_i = 2$ there is good confidence in accepting h'_i and so until level 4 with the maximum confidence. The cls are represented with radar plots. Hypothesis testing and other statistics are applied by DIAGXTRM to evaluate execution time patterns and other characteristics that traces can exhibit. DIAGXTRM is available at `https://forge.onera.fr/projects/diagxtrm2`.

3 EVT & Dependences

In this section we detail the effects that statistical dependence of measurements has on pWCET estimates.

The pWCET estimate in case of extremal independence $\overline{\mathcal{C}}^{ei}$ is greater than or equal to The pWCET estimate in case of independence $\overline{\mathcal{C}}^{i}$: $\mathsf{icdf}_{\overline{\mathcal{C}}^{ei}} \geq \mathsf{icdf}_{\overline{\mathcal{C}}^{i}}$, [10]. The partial ordering between $\overline{\mathcal{C}}^{ei}$ and $\overline{\mathcal{C}}^{i}$ is ensured if and only if both $\overline{\mathcal{C}}^{ei}$ and $\overline{\mathcal{C}}^{i}$ follow the same average distribution $\mathcal{C}$.

The inequality states that dependence, up to a certain degree which assure EVT applicability (local dependence and extreme dependence $h_{2,1} \wedge h_{2,2}$), provides more pessimistic pWCET estimates than the independence. In other words, clusters of execution times within the trace (short range dependences) or patterns between far away measurements (extremal dependences) make rare events more probable. The EVT accounts for that with more pessimistic pWCET i.e. more conservative pWCET estimates, and the pWCET with dependence remains a safe modeling of the task worst-case.

We remark that the former condition poses some issues to works that aim at creating independence between measurements, [19, 10]. Techniques like re-sampling and de-clustering that are normally applied in some domains, have to be thoroughly investigated before being applied to the pWCET problem because they can produce optimistic pWCETs.

Trace $\mathcal{T}_{\overline{\mathcal{C}}}$ cannot be changed if we aim at guaranteeing safe pWCET. The representativity of the measurements in characterizing the actual task behavior with dependence effects cannot be modified. Eventual changes to assure full statistical independence between measurements and "better-to-apply-EVT" could produce optimistic pWCETs. The lost of representativity of the measurements could end up into unsafe pWCET estimates.

Dependence Case Study. Some test cases are applied for validating the dependence impact on pWCET estimates we propose; each test case is represented by a trace of execution time measurements. The execution times which are measured with the tools available for the test cases, are all in CPU cycles.

- *trace1* is a trace of execution time measurements from an industrial avionic safety-critical multi-core system, [24]; the task under observation executes on one core while another core is doing interfering I/O activities. In its original version (V1), *trace1* has weak extreme independence; from V1 we obtain a modified version of *trace1*, V2 by randomly re-sampling the measurements; V2 has stronger extreme independence than V1.
- *trace2* is a trace of execution time from the *dijkstra* task of the TACLeBench[5] executing on a Kalray many-core platform, [21]. The task under observation executes on one core, while other cores produce interference through shared memory. In its original version V1,

[5] `http://www.tacle.eu/index.php/activities/taclebench`.

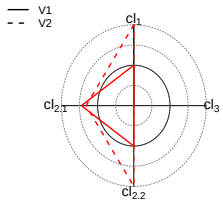


Figure 6 Radar plot for *trace1* with two degrees of extreme independence applied.

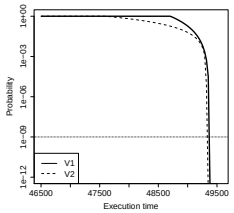


Figure 7 CCDF representation of pWCETs from PoT EVT applied to *trace1*.

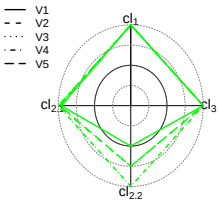


Figure 8 Radar plot of *trace2*: pWCET confidence of the 5 versions for *trace2*.

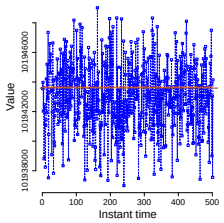


Figure 9 Portion of trace of execution time measurements for *trace2* V1.

trace2 has weak extreme independence. We define 4 more versions of *trace2* by randomly re-sampling the measurements; intermediate versions V2, V3 and V4 have decreasing degree of extreme dependence (more extreme independence); V5 has full independence between measurements.

- *trace3* and *trace4* are artificial traces of execution time measurements extracted from a Gaussian distribution. To *trace3* there has been added local dependence between measurements to reproduce the effect of non-time randomized system elements to the task execution behavior e.g., locality effects from caches. To *trace4* there has been added extremal dependence between measurements to reproduce the effect of periodic inputs. *trace3* has been modified from its original version V1 with local dependence with 4 more versions. V5 has full independence obtained by randomly re-sampling measurements in *trace3* while intermediate versions V2, V3 and V4 have decreasing degree of dependence. *trace4* has been modified from its original version V1 with extreme dependence by randomly re-sampling the measurements; V2 is the least re-sampled version of *trace4* with still strong dependence, V3 is more re-sampled than V2 and V4 is an even more re-sampled version of *trace4*.

The proposed case study is a small fraction of the benchmarks investigated; it is representative because the composing traces exhibit dependence and through which it is possible to illustrate the effects of artificially induced independence on the safety of the pWCETs. The traces are processed with DIAGXTRM for deriving safe and confident pWCETs; all of them and more, except the industrial one, are available at `https://forge.onera.fr/projects/diagxtrm2`.

Dependence Results. Figure 6 details the confidence level of *trace1* in its original version V1 and for the modified version V2; $cl_{2.2}$ increases from V1 to V2 as result of the artificially induced independence on the measurements. Figure 7 illustrates the effect of extreme independence on the pWCET estimates: *by increasing the independence the pWCET estimate decreases*.

To *trace1* the EVT is not applicable i.e. the matching hypothesis fails with $cl_3 = 0$; nonetheless, it helps us to understand the impact of extremal independence on pWCET estimates. Artificially inducted independence, even if with small effects as for *trace1*, acts reducing the pWCET estimates. *If the representativity of a system with dependence is not guaranteed in case of dependent measurements, the risk is to have unsafe pWCET estimates.*

Figure 8 details the radar plot for the EVT approach applied to *trace2*. The confidence levels for the 5 versions of *trace2* are represented and tells that the EVT is confidently applicable to all the versions, included those with certain degree of dependence; $cl_{2.2}$ increases from V1 to V5, meaning that the random re-sampling applied is able to break the extreme

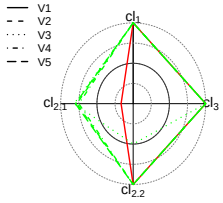


Figure 10 Radar plot of *trace3*: confidence levels for the 5 versions of *trace3*.

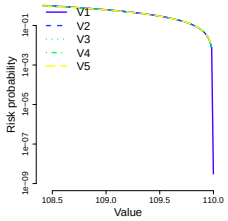


Figure 11 CCDF representation of pWCETs from PoT EVT applied to *trace3* and its modified versions.

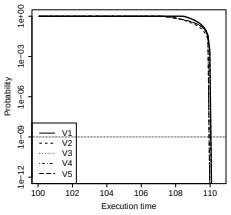


Figure 12 CCDF representation of pWCETs from PoT EVT applied to *trace3* and its modified versions.

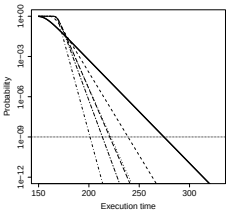


Figure 13 CCDF representation of pWCETs from BM EVT applied with increasing block size to *trace3*.

dependence existing within *trace2* V1. For *trace2*, the pWCET variations due to artificial induced independence are negligibly small with respect to the dependent case V1: there is no impact on the pWCET of extreme dependence estimates for *trace2*. It is a particular case, due to the trace small variability and the shape of the resulting pWCET.

Figure 9 illustrates a portion of the trace of measurements for *trace2* V1; although no particular input or execution condition are exercised (dijkstra task in a many-core execution with interference generated from a concurrent task with no particular input imposed, [21]), the behavior of the peaks over the threshold selected (horizontal line) could follow a oscillatory periodic pattern, hence some degree of extreme dependence exists. Future work will focus on defining possible patterns with statistics.

Figure 11 details the PoT EVT approach applied to *trace3*. The pWCET variations of the version V2, V3, V4 and V5 are small with respect to the dependent case V1, but they all act reducing the pWCET estimates. The guarantee of having safe pWCETs from artificially independent traces reduces because the trace modifications are not conservative. Figure 10 illustrates the radar plot of the confidence levels for the 5 versions of *trace3*: the EVT is confidently applicable to all the versions, including those with certain degree of dependence.

We hereby specialize the comparison between PoT and BM, proposed first in [23], to the statistical dependence case. The BM EVT is applicable to more dependent cases than the PoT, since grouping consecutive measurements into blocks and selecting only the maximum of each block would break possible local dependences and extremal dependences. We say that the BM EVT is more robust with respect to dependence (local dependence and/or extremal dependences) than the PoT because of the capacity of block maxima of filtering measurements. Also, the PoT filters measurements i.e. those below the threshold, but it does not with respect to dependences. The problem with BM is that filtering dependences with large block sizes would reduce the impact of dependence on the rare events, thus resulting into possibly optimistic pWCET estimates. Figure 13 details the BM EVT applied to *trace4* with different block sizes. The continuous line pWCET is for block size of 10 measurements, the dotted line for block size of 20, and so on. The more the block size increases and the dependence between block maxima decreases, the more the pWCET estimates decreases augmenting the risk of optimistic pWCET estimates.

Figure 12 details the PoT EVT approach applied to *trace4*. The pWCET variations induced by less dependence (versions V2 to V5) are small with respect to V1, but act reducing the pWCET estimates questioning the safety of pWCET estimates with artificial independent traces.

By comparing BM and PoT with respect to dependence effects, we observe that the PoT is a more robust approach than BM to independence effects (artificially created independence).

Nonetheless, with neither of them it is possible to guarantee safe pWCET estimates if the traces loose their representativity with artificially reduced dependence. We are currently employing robustness as intuitive notion, future work will be devoted to formalize it for the EVT.

Forcing independence into dependent execution time measurements makes the trace not representative anymore of the dependent system behavior. As a result, the pWCET could end up into a non safe anymore worst-case estimation of the task execution behavior.

4 EVT & Multi-Mode Tasks

The pWCET estimate depends on the execution conditions applied for the measurements; the EVT is able to produce the worst-case bound for that condition only.

Measurement-based timing analysis, either probabilistic MBPTA or deterministic [15], have to cover every possible execution condition and inputs in order to guarantee the absolute worst-case bound estimate. Thus, *the measurements have to be representative of all the possible execution conditions the system can experience.*

With $J = \{j\}$ the finite set of possible measurement execution conditions for a system, there exist two ways of integrating all the scenarios into the MBPTA:
Trace-merging consists of merging all the traces $\mathcal{T}_{\mathcal{C}^j}\ \forall j \in J$ within a unique trace $\mathcal{T}_{\mathcal{C}}$, $\mathcal{T}_{\mathcal{C}} \stackrel{def}{=} \bigcup_{j\in J} \mathcal{T}_{\mathcal{C}^j}$; the EVT is applied to $\mathcal{T}_{\mathcal{C}}$ for deriving $\overline{\mathcal{C}}$ as the worst-case distribution for J.
Envelope consists of applying the EVT to each measurement condition j and get $\overline{\mathcal{C}}^j$ for all $j \in J$. The worst-case distribution $\overline{\mathcal{C}}$ that upper bounds every $j \in J$ is such that: $\overline{\mathcal{C}} \stackrel{def}{=} max_{j\in J}\{\overline{\mathcal{C}}^j\}$ and $\mathsf{icdf}_{\overline{\mathcal{C}}}(C) \stackrel{def}{=} max_{j\in J}\{\mathsf{icdf}_{\overline{\mathcal{C}}^j}(C)\}$.

Measurements representativity with respect to the system behaviors needs the measurements to include every possible execution behavior for the task. Instead of enumerating all the possible execution conditions for a system, the dominance between some of them ($\mathsf{icdf}_{\overline{\mathcal{C}}^1} \geq \mathsf{icdf}_{\overline{\mathcal{C}}^2}$) and the knowledge of worst conditions would keep the measurements representative and allow for worst/safe pWCET estimates. Both trace-merging and envelope approaches rely on knowing all the measurement conditions.

Task inputs and operational modes contribute to define the execution conditions of the system and its tasks. Hence, the behavior of multi-mode tasks is assimilated to multiple execution conditions; trace-merging and envelope are the approaches that can be applied to multi-mode tasks for guaranteeing worst pWCET and measurement representativity. Figure 4 presents a trace of execution time measurements for a two-mode task, while Figure 5 presents the three-mode task case.

Multi-Mode Case Study. Some test cases are applied for validating the multi-mode task study we propose; each test case is represented by a trace of execution times measurements. The traces are representative of actual execution conditions; the execution times, which are measured using the tools available for the test cases, are all in CPU cycles.

- *trace5* is a trace from an industrial avionic safety-critical embedded system (different than *trace1*) where the task is a two-mode application executing on a multi-core platform. The task under observation executes on one core, while another core is doing interfering I/O activities, [24];
- *trace6* comes from the *ns* Mälardalen benchmark task implementation[6] executed in

[6] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

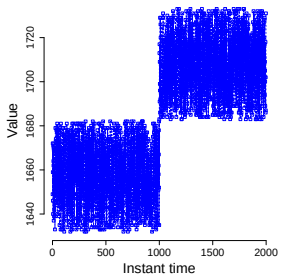


Figure 14 *trace5* with two operational modes.

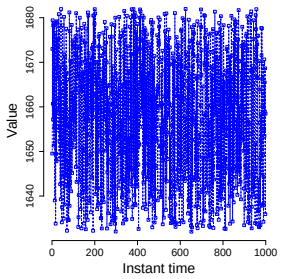


Figure 15 *trace5* with only the first mode represented.

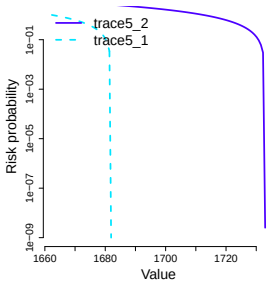


Figure 16 CCDF representation of *trace5* decomposed into *trace5_1* and *trace5_2*.

isolation on a multi-core real-time system, [12]. *trace6* describes a four-mode task where the 4 inputs (for 4 task paths) are randomly picked at runtime;

- *trace7* is a trace from a FPGA implementation of a multi-core real-time system; it is obtained from the *lms* task of the Malardalen benchmark which trace has been obtained by running the task under observation with interference from other tasks, [7]. The task trace results into a two operational mode of execution times not controlled whatsoever (no particular input exercised), and the measurements capture the two different modes with their effects on the execution time.

Among the possible benchmarks investigated, we report few representative cases of multi-mode tasks. The traces are processed with DIAGXTRM; all of these, except the industrial one, are available at `https://forge.onera.fr/projects/diagxtrm2`.

Multi-Mode Results. Figure 14 illustrates the two-mode task represented by *trace5*; Figure 15 details the first part of the trace *trace5_1*, which describe task execution mode 1. In order to apply the EVT, *trace5* has to be decomposed into two traces *trace5_1* and *trace5_2*, respectively for the first mode and for the second mode. Only the envelope approach can be used with *trace5* and the reason is that without decomposing *trace5* into two traces, h'_1 and h'_3 cannot be verified since the peaks above the threshold would belong to both modes. Figure 16 details the pWCETs from the two parts of *trace5*, with *trace5_2* dominating *trace5_1*. For *trace5* if is sufficient to have measurements representative of the worst mode to guarantee the worst pWCET.

The two-mode *trace6* is illustrated in Figure 17; it is possible to apply the EVT to *trace6* with the trace-merging approach, because the peak above the threshold would belong only to the mode with larger execution times (worst mode). With that, both h'_1 and h'_3 can be confidently verified. The envelope and the trace-merging approaches produces same pWCETs for *ns* (*trace6*) because the worst-case condition dominates all the others. Figure 18 depicts the pWCET estimate and the perfect fit between the pWCET distribution and the measured execution time peaks (best fitting the input measurements is a critical element for the EVT application).

The two-mode *trace7* is detailed in Figure 19. In order to apply the EVT, *trace7* has to be decomposed into *trace7_1* and *trace7_2*, respectively for the first mode and for the second mode. Only the envelope approach can be used with *trace7* and the two separated traces. Figure 20 details the pWCETs from the two parts of *trace7*. The worst pWCET is the maximum of the two pWCET because none of the modes dominates; the knowledge of both modes (representativity) is essential in order to conclude about the worst pWCET.

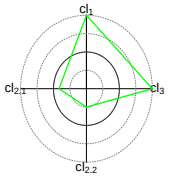


Figure 17 Radar plot for *trace6*.

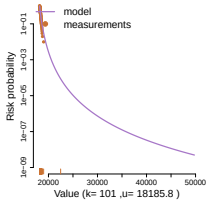


Figure 18 CCDF representation of *trace6* pWCET.

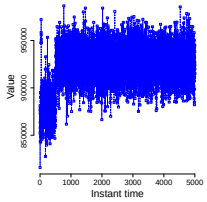


Figure 19 *trace7* with two operational modes.

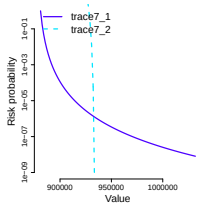


Figure 20 CCDF representation of *trace7* decomposed into *trace7_1* and *trace7_2*.

As a marginal note, we observe that DIAGXTRM infers two different shapes for the pWCET distributions of *trace7_1* and *trace7_2*; this is because DIAGXTRM applies a best fit procedure to the input measurements, and in [24] it has been demonstrated that only the best fit guarantees safe pWCET estimates. Only the best fit allows respecting the representativity of measurements and seeking for the best pWCET shape to cope with that.

When the EVT is not applicable to the full trace (trace-merging) i.e. *trace5* and *trace7*, the trace has to be decomposed into sub traces each characterizing a task mode; the representativity has to be preserved by not neglecting any sub trace/mode. Then, the EVT can applied to all the sub traces (envelope) and with the guarantee of having inferred the worst pWCET estimate.

5 Conclusions

MBPTA and EVT demand for representative trace of execution time measurements in order to provide safe and confident pWCET estimates.

In case of statistical dependence, changing execution time measurements and artificially create independence in order to have "better EVT applications" may cause the effect of reducing the safety of the pWCET estimates; this is not affordable with worst-case execution time estimates. The only allowed modifications to execution time traces are the conservative ones. Instead, with multi-mode tasks and multiple execution conditions, representative measurements have to include all the execution conditions in order to be able to infer the worst pWCET; execution conditions cannot be neglected, especially worst-case conditions.

Full coverage of system execution conditions and the dominance between execution conditions will be thoughtfully investigated in future work. The representativity of the measurements will be quantified including also the identical distribution hypothesis and other system parameters. Statistic metrics will be developed to identify pattern and execution behavior for real-time task. Moreover, the differences between BM EVT and PoT EVT will continue to be investigated. Special attention will be given to measurements robustness, measurements representativity and trace changes.

References

1 J. Abella, J. del Castillo, M. Padilla, and F. J. Cazorla. Extreme value theory in computer sciences: The case of embedded safety-critical systems. In *6th International Conference on Risk Analysis (ICRA)*, 2015.

2 Jaume Abella, Eduardo Quiñones, Franck Wartel, Tullio Vardanega, and Francisco J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in MBPTA. In *26th Euromicro Conference on Real-Time Systems, (ECRTS)*, 2014.

3 W. A. Brock, J. A. Scheinkman, W. D. Dechert, and B. LeBaron. A Test for Independence based on the Correlation Dimension. *Econometric Reviews*, 15(3):197–235, 1996.

4 Alan Burns and Stewart Edgar. Predicting computation time for advanced processor architectures. In *12th Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

5 Francisco J. Cazorla, Tullio Vardanega, Eduardo Quinones, and Jaume Abella. Upper-bounding program execution time with extreme value theory. In *WCET*, 2013.

6 L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *23nd IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

7 Corentin Damman, Gregory Edison, Fabrice Guet, Eric Noulard, Luca Santinelli, and Jerome Hugues. Architectural performance analysis of FPGA synthesized LEON processors. In *Proceedings of the IEEE International Symposium on Rapid System Prototyping*, 2016.

8 J. L. Díaz, D. F. Garcia, K. Kim, C. G. Lee, L. L. Bello, J. M. López, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *23rd of the IEEE Real-Time Systems Symposium (RTSS)*, 2002.

9 P. Embrechts, C. Klüppelberg, and T. Mikosch. *Modelling extremal events for insurance and finance*. Applications of mathematics. Springer, Berlin, Heidelberg, New York, 1997.

10 C. A. T. Ferro and J. Segers. Automatic Declustering of Extreme Values Via an Estimator for the Extremal Index. *Technical Report*, 2002.

11 M. K. Gardner. *Probabilistic analysis and scheduling of critical soft real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. AAI9953022.

12 F. Guet, L. Santinelli, and J. Morio. On the reliability of the probabilistic worst-case execution time estimates. In *8th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2016.

13 D. Kwiatkowski, P. C. B. Phillips, P. Schmidt, and Y. Shin. Testing the null hypothesis of stationarity against the alternative of a unit root : How sure are we that economic time series have a unit root? *Journal of Econometrics*, 54(1-3):159–178, 00 1992.

14 F. Laio. Cramer-von Mises and Anderson-Darling goodness of fit tests for extreme value distributions with unknown parameters. *Water Resources Research*, 40, 2004.

15 Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems*, 2016.

16 M. R. Leadbetter. Extremes and local dependence in stationary sequences. *Stochastic Processes and their Applications*, 35, 1983.

17 M. R. Leadbetter. On a basis for peaks over threshold modeling. *Statistics & Probability Letters*, 12(4):357–362, 1991.

18 George Lima, Dario Dias, and Edna Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *28th Euromicro Conference on Real-Time Systems, (ECRTS)*, 2016.

19 C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

20 Cristian Maxim, Adriana Gogonel, Irina Asavoae, Mihail Asavoae, Liliana Cucu-Grosjean, and Walid Talaboulma. Reproducibility and representativity – mandatory properties for the compositionality of measurement-based WCET estimation approaches. In *The 9th International Workshop on Compositional Theory and Technology for Real-Time Embedded System (CRTS)*, 2016.

21 Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. The variability of application execution times on a multi-core platform. In *16th International Workshop on Worst-Case Execution Time Analysis, 2016*, pages 6:1–6:11, 2016.

22 R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time COTS based systems. In *Real-Time Systems Symposium. (RTSS). 28th IEEE International*, 2007.

23 L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the Sustainability of the Extreme Value Theory for WCET Estimation. In *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 21–30, 2014.

24 Luca Santinelli, Fabrice Guet, and Jerome Morio. Revising measurement-based probabilistic timing analysis. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017.

Tightening the Bounds on Cache-Related Preemption Delay in Fixed Preemption Point Scheduling*

Filip Marković[1], Jan Carlson[2], and Radu Dobrin[3]

1 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`filip.markovic@mdh.se`
2 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`jan.carlson@mdh.se`
3 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`radu.dobrin@mdh.se`

Abstract

Limited Preemptive Fixed Preemption Point scheduling (LP-FPP) has the ability to decrease and control the preemption-related overheads in the real-time task systems, compared to other limited or fully preemptive scheduling approaches. However, existing methods for computing the preemption overheads in LP-FPP systems rely on over-approximation of the evicting cache blocks (ECB) calculations, potentially leading to pessimistic schedulability analysis.

In this paper, we propose a novel method for preemption cost calculation that exploits the benefits of the LP-FPP task model both at the scheduling and cache analysis level. The method identifies certain infeasible preemption combinations, based on analysis on the scheduling level, and combines it with cache analysis information into a constraint problem from which less pessimistic upper bounds on cache-related preemption delays (CRPD) can be derived.

The evaluation results indicate that our proposed method has the potential to significantly reduce the upper bound on CRPD, by up to 50% in our experiments, compared to the existing over-approximating calculations of the eviction scenarios.

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

Keywords and phrases Real-time systems, CRPD Analysis, WCET analysis, Limited Preemptive Scheduling, Fixed Preemption Point Approach

Digital Object Identifier 10.4230/OASIcs.WCET.2017.4

1 Introduction

Preemption-related overheads are of significant importance in real-time scheduling, as they may have a decisive impact on the overall system schedulability. It has been shown that, in some cases, the cumulative preemption overhead may increase a task's execution time up to 33% [11].

In this context, the Limited Preemptive Scheduling (LPS) paradigm has emerged as a valuable scheduling approach in order to reduce the overall preemption overhead of such

* We want to express our gratitude to the EUROWEB+ project that partially funded this work.

systems, and, consequently, to increase their schedulability. In this paper, within LPS we consider Fixed Preemption Points (LP-FPP) because it provides more predictability with respect to the calculation of the preemption-related overhead, compared to Preemption Thresholds Scheduling and Deferred Preemption Scheduling. The dominance considering the overhead calculation predictability in LP-FPP comes from the fact that the preemption points are selected off-line, during the design phase of the system, unlike the other two LPS approaches where the preemption points are known only at runtime.

In order to calculate the preemption-related overhead we need to consider different types of costs such as: scheduling cost, pipeline cost, bus-related cost, et cetera. However, the cache-related preemption delay (CRPD) is often the largest part of the preemption-related overhead [3], and it denotes the time that is needed for reloading the cache lines evicted by the preempting task.

The basic CRPD calculation needs to consider the following factors: (1) *The point $\mathcal{P}$ in the code of the preempted task where the preemption occurs*; (2) *Cache blocks used until $\mathcal{P}$ that may be evicted by one or more higher priority (preempting) tasks and reused afterwards in the remaining execution of the preempted task*; and finally (3) *The evicting cache blocks of the preempting tasks.* Considering this knowledge, we can see that the major difference between fully preemptive and LP-FPP scheduling comes from the fact that in LP-FPP a preemption point selection defines the non-preemptive regions between consecutive points and consequently affects the set of useful cache blocks for each selected preemption point. Contrary, in a fully preemptive scheduling we cannot control the number of preemption points and useful cache blocks per preemption point, therefore the CRPD value is defined per task. Despite the significant research done for the CRPD computation under fully preemptive scheduling, e.g., [1, 2, 10], relatively little work is done considering the LP-FPP approach. This led to the bifurcation in the research between scheduling and CRPD analysis, that is visible in the fact that a majority of the papers considering LP-FPP schedulability tests e.g., [4, 12] assumed that the upper bound CRPD values are provided for each preemption point. Such model for CRPDs introduces a considerable overestimation of a cumulative CRPD value per task, especially in the sporadic task model where we need to assume that the preempting tasks may be released at any time instant after the minimum inter-arrival time from their previous deadline. Consequently, we need to assume that each preemption point may experience the worst case eviction scenarios which is often not feasible for all preemption points of a task.

Recently, Cavicchio et al. [8] proposed the CRPD computation for the LP-FPP approach that considers the CRPD for each pair of adjacent preemption points, and it significantly improves the CRPD estimation under LP-FPP. However, the assumption about the evicting cache blocks from preempting tasks still remains an overestimation as it assumes that each pair of adjacent preemption points is affected by all higher priority tasks.

In this paper, we propose the methodology for tightening the upper bounds on the CPRD values in sporadic task sets, scheduled under the Fixed Priority LP-FPP paradigm, by using a less pessimistic calculation of the evicting cache blocks of the preempting tasks. In our proposed approach, we calculate a single CRPD value per preempted task by integrating the scheduling- and cache level analysis to identify infeasible preemptions and further formulate a constraint satisfaction models. The preliminary evaluation results indicate that our methodology can significantly reduce the upper bound of CRPD values up to 50% compared to the existing methods for ECB calculation.

The remainder of the paper is organised as follows: In Section 2 we introduce the system model, terminology and notations used in the paper which describe the task and cache model.

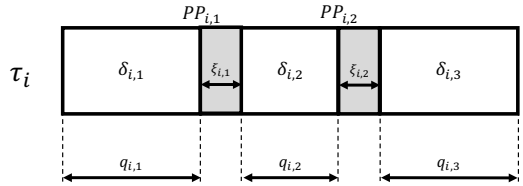


Figure 1 Task model with two selected preemption points ($PP_{i,1}$ and $PP_{i,2}$ with respective CRPD overheads $\xi_{i,1}$ and $\xi_{i,2}$) which form three non-preemptive regions ($\delta_{i,1}$, $\delta_{i,2}$ and $\delta_{i,3}$ with respective worst case execution times $q_{i,1}$, $q_{i,2}$ and $q_{i,3}$)

In Section 3 we describe the related work and problem formulation of the paper. In Section 4 we describe the proposed methodology for tightening the upper CRPD bound in the LP-FPP task model. In Section 5 we discuss the preliminary evaluation of the methodology and Section 6 concludes the paper.

2 System Model

We consider a real-time sporadic task model Γ composed of n tasks τ_i $(1 \leq i \leq n)$ scheduled under the Fixed Priority (FP) paradigm on a single processor. Tasks are ordered in a decreasing priority order (and each task τ_i generates an infinite number of task instances $\tau_{i,j}$). Each task is described with a tuple $\{P_i, C_i^{np}, D_i, T_i\}$ where P_i denotes the task's priority, C_i^{np} denotes the non-preemptive worst case execution time of $\tau_{i,j}$. Considering each task instance, D_i denotes a relative deadline, and the minimum inter-arrival time between two consecutive instances of τ_i is denoted with T_i.

We also consider the LP-FPP approach and therefore assume the sequential task splitting model such that each task is divided by d selected preemption points $PP_{i,k}$ $(1 \leq k \leq d)$ with an estimated preemption-related overhead value $\xi_{i,k}$. Furthermore, selection points form $d+1$ non-preemptive regions $\delta_{i,k}$ with worst case execution times $q_{i,k}$ such that $\sum_{k=1}^{d+1} q_{i,k} = C_i^{np}$ (see Figure 1).

We furthermore extend the system model in order to consider preemption-related overhead calculation assuming a direct-mapped cache.

As previously mentioned, the preemption-related overhead is primarily caused by the cache-related preemption delay (CRPD) compared to the other cost types such as scheduling cost, pipeline cost etc., which we consider as constants that are already included in the non-preemptive execution time C_i^{np} of a task τ_i. We denote the upper bound on CRPD of a task τ_i with γ_i and furthermore we denote with C_i^{γ} the worst case execution time considering preemptions on τ_i such that $C_i^{\gamma} = C_i^{np} + \gamma_i$.

As γ_i represents the time needed for reloading the cache blocks of the preempted task τ_i that are evicted by the preempting tasks τ_h, it is bounded by the following equation $\gamma_i = g \times BRT$ where g is the upper bound on the number of needed cache block reloads caused by preemptions, precisely evictions of cache blocks belonging to τ_i, and BRT is a cache block reload time. Furthermore, in order to calculate g we define two concepts of cache blocks considering the preemptions:

Useful cache block (UCB) – As proposed by Lee et al. [10], UCB at preemption point $PP_{i,k}$ is a memory block m of the preempted task such that: a) m may be cached at $PP_{i,k}$, and b) m may be reached from $PP_{i,k}$ and reused at program point $\mathcal{P}$ that is reachable from $PP_{i,k}$ without eviction of m on this path. If the preemption occurs at $PP_{i,k}$ we need to address only the memory blocks that are cached and may be reused. More formally,

considering a non-preemptive region $\delta_{i,k}$ we define a set of useful cache blocks $UCB_{i,k}$ such that $m \in UCB_{i,k}$ if and only if non-preemptive region $\delta_{i,k}$ has m as a useful cache block at point $PP_{i,k}$. Furthermore, we define the set of useful cache blocks UCB_i per τ_i: $UCB_i = \bigcup_{\delta_{i,k} \in \tau_i} UCB_{i,k}$ $(1 \leq k \leq d)$. Notice that we do not consider the useful cache blocks of the last non-preemptive region $\delta_{i,d+1}$ as it cannot be preempted.

Evicting cache block (*ECB*) – defines a memory block m of the preempting task that may be accessed during the execution of the preempting task.

In this paper we also consider the evicting cache set ECB_i of a task τ_i such that a memory block $m \in ECB_i$ if and only if τ_i may evict cache block m.

Finally, in some cases we consider the upper bound on the single preemption cost $\xi_{i,k}$ at preemption point $PP_{i,k}$ which is computed considering the useful memory blocks $UCB_{i,k}$ and the union of the evicting cache blocks ECB_h from all possibly preempting tasks:

$$\xi_{i,k} = \left| UCB_{i,k} \cap \left(\bigcup_{h \in hp(i)} ECB_h \right) \right| \times BRT \,.$$

3 Related Work

In the seminal paper that considered preemption cost aware schedulabity tests, Busquets et al. [6] considered an upper bound on the preemption cost of a single job of a preempting task τ_h that executes during the response time of a preempted task τ_i, which is defined by a term $\gamma_{i,h}$. This value is calculated for each higher priority job that may be released during the response time. Later, Petters et al. [13] proposed the more precise analysis considering the upper bound on the preemption cost of all jobs of τ_h executing during the response time of τ_i. By using this approach they were able to reduce the pessimism from the previously proposed method that considered the upper bound on preemption cost of each τ_h's job separately. Considering exact $\gamma_{i,h}$ computation there are two dominant approaches: UCB-union and ECB-union, and both of them assume that the evicting cache blocks of the preempting jobs are given and will definitely evict intersected useful cache blocks of the preempted task. However, those approaches do not account for the fact that the additional preemptions may result in a smaller preemption costs than the prior accounted ones. This fact was accounted by introducing the multiset-based computation of CRPD, proposed by Staschulat et al. [15] with a drawback that it over-estimates the number of preemptions that have an impact on the response time and does not account for the variability in the number of possible intermediate preemptions. Therefore, Altmeyer et al. [1] proposed ECB/UCB-Union Multiset approaches, which account for the precise number of possible intermediate preemptions.

Considering periodic task systems, Ramaprasad and Miller [14] analysed the feasible preemptions by analysing each job of a preempted task throughout the hyper-period considering the worst-case placement of preemptions of preempting tasks. In this paper, they considered that not every preempting job of τ_h can cause a preemption on the preempted task τ_i.

In the LP-FPP approach, the majority of the papers defining the schedulability tests e.g., [4, 7, 12] assume the upper bound on preemption cost values for each preemption point. In sporadic task systems this approach can lead to a huge overestimation of the preemption cost because the release time of the higher priority job is not definitely known, thus we would need to assume that each point suffers from the worst case eviction scenario. Improving the assumption of per point preemption cost, Cavicchio et al. [8] proposed the CRPD computation for the LP-FPP approach that considers the CRPD for each pair of adjacent preemption points which significantly improved the CRPD estimation under LP-FPP. However, the assumption about the evicting cache blocks from preempting tasks still remains

Data: Task set Γ
Result: Set of tightened preemption overhead values for each task from Γ

1 **for** $i \leftarrow 2$ **to** n **do**
2 $V \leftarrow$ Generate a set of variables that represent a preemption affection by τ_h at $PP_{i,k}$ of τ_i
3 $C \leftarrow$ Generate a set of constraints that define infeasible preemption combinations
4 $G \leftarrow$ Generate a goal function for the given constraint problem
5 $\gamma_i \leftarrow$ compute CRPD of τ_i by solving the constraint problem defined by V, C and G
6 $C_i^{\gamma} \leftarrow C_i^{np} + \gamma_i$
7 **end**

Algorithm 1: Algorithm for tightening the upper bound of CRPD in a taskset.

an overestimation as it assumes that each pair of adjacent preemption points is affected by all higher priority tasks.

Contrasting these methods, we propose an improved method for tightening the bounds on the preemption cost calculations in sporadic LP-FPP task scheduling, by identifying infeasible eviction scenarios.

4 Computing CRPD bounds

In order to reduce the CRPD overestimation we propose a method that for each preempted task τ_i in a taskset first identifies infeasible preemption combinations, and then calculates the maximum CRPD considering the remaining preemption combinations. Concretely, the basic idea of the method is to identify cases where two instances of a higher priority task τ_h cannot affect the preemption costs at two preemption points of the same τ_i instance.

Considering the taskset Γ, we propose an algorithm that computes a CRPD upper bound γ_i for each task in a decreasing priority order (see Algorithm 1).

For each task, we first (line 2) generate a set of variables, each representing the case when a certain higher priority task τ_h affects the preemption cost of a certain preemption point $PP_{i,k}$ of the considered task τ_i. Next (line 3), we generate a set of constraints that capture identified infeasible preemption combinations, and a goal function (line 4) representing the preemption cost associated with the different preemption scenarios. Furthermore (line 5), we compute the upper-bounded cache-related cost γ_i of the preempting task τ_i by solving the constraint problem defined by V, C and G. Next, we compute the execution time of τ_i considering the CRPD upper bound C_i^{γ} (line 6), to be used in the following iterations. This is done because the preemptive execution time of the higher priority task may impact the CRPD computation of the lower priority task and because we need to address the case of nested preemptions.

We describe in detail each of the computations in the following subsections. To illustrate the approach, we will use the tasks depicted in Figure 2 as a running example, focusing on the second iteration of the algorithm, meaning that the preempted task τ_3 is being considered.

In the example, we also consider the preempting tasks (τ_1 and τ_2). Furthermore, τ_1 has the worst case execution time $C_1 = 30$ and the minimum inter-arrival time $T_1 = 100$. Its set of evicting cache blocks ECB_1 consists of two cache blocks presented by integers 1 and 2. The preempted task τ_3 is divided by three selected preemption points ($PP_{3,1}$, $PP_{3,2}$ and $PP_{3,2}$) and it consists of four non-preemptive regions with given WCET values ($q_{3,1} = 20$, $q_{3,2} = 10$, etc.). Set of useful cache blocks, e.g., $UCB_{3,1}$, is defined for each preemption point.

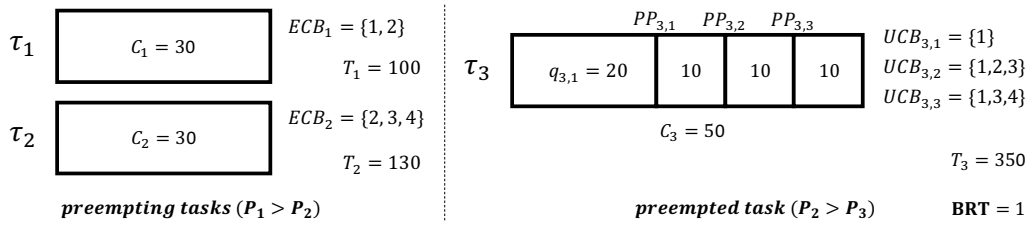


Figure 2 Running example: Two higher priority tasks τ_1 and τ_2 and the preempted task τ_3.

4.1 Variable declaration

Considering a preemption point $PP_{i,k}$ of a preempted task τ_i and a single preempting task τ_h, we declare a boolean variable $X_{h,k}$ which represents the case when τ_i is preempted at $PP_{i,k}$ and the associated preemption cost $\xi_{i,k}$ is affected by an instance of τ_h. Consequently, for τ_i we generate $d \times (i-1)$ boolean variables $X_{h,k}$ where d is the number of preemption points of τ_i, since $(i-1)$ is the number of tasks with a higher priority than τ_i. Formally, the set V of generated variable declarations is defined as:

$$V = \left\{ X_{h,k} \in \{0,1\} \;\middle|\; (1 \leq h < i) \wedge (1 \leq k \leq d) \right\}.$$

The set of variable declarations for τ_3 in the running example is:

$$V = \left\{ X_{1,1} \in \{0,1\},\ X_{1,2} \in \{0,1\},\ X_{1,3} \in \{0,1\},\ X_{2,1} \in \{0,1\},\ X_{2,2} \in \{0,1\},\ X_{2,3} \in \{0,1\} \right\}.$$

4.2 Constraint formulation

Considering the constraints, we are interested in identifying infeasible preemption combinations, focusing on preemption scenarios where two instances of a preempting task τ_h can directly affect the cost of only one of the two preemption points $PP_{i,k}$ and $PP_{i,l}$ $(k < l \leq d)$ of a preempted task τ_i, but not both.

To formally define the constraint generation, we first define the computation of the maximum time interval $I_i^{k,l}$ from the start time of $\delta_{i,k}$ until the start time of $\delta_{i,l+1}$. Informally, $I_i^{k,l}$ represents the time interval during which the two instances of τ_h must be released in order to affect both preemption points.

The definition of $I_i^{k,l}$ is based on the traditional response time analysis, but considering only a subset of the task, from $\delta_{i,k}$ to $\delta_{i,l}$. The longest interval is found by assuming that the higher priority tasks arrive as early and as often as possible in the considered interval. Since there is no possibility of blocking by lower priority tasks in this case, we only sum the execution time of the non-preemptive regions with respective upper-bounded preemption costs (from k to l) and the interference from the higher priority tasks. Each upper-bounded preemption cost $\xi_{i,w}$ is added to the respective execution time $q_{i,w}$ of the non-preemptive region $\delta_{i,w}$ in order to account for the impact of the CRPD on the length of the non-preemptive region. The iterative computation of $I_i^{k,l}$ is defined as follows:

$$\begin{cases} I_i^{k,l}(0) = \sum_{w=k}^{l} (q_{i,w} + \xi_{i,w}) + \sum_{h \in hp(\tau_i)} C_h^{\gamma}, \\ I_i^{k,l}(z) = \sum_{w=k}^{l} (q_{i,w} + \xi_{i,w}) + \sum_{h \in hp(\tau_i)} \left(\left\lfloor \frac{I_i^{k,l}(z-1)}{T_h} \right\rfloor + 1 \right) C_h^{\gamma}. \end{cases}$$

Finally, we define $I_i^{k,l}$ to be the least fixed point of the recursion, i.e., $I_i^{k,l} = I_i^{k,l}(z)$ where z is the lowest value for which $I_i^{k,l}(z) = I_i^{k,l}(z+1)$.

Next, we show how the time interval $I_i^{k,l}$ can be used to identify infeasible preemption combinations.

▶ **Lemma 1.** *If task τ_h affects the preemption cost at $PP_{i,k}$, then an instance of τ_h was released between the start of $\delta_{i,k}$ and the start of $\delta_{i,k+1}$.*

Proof. An instance of τ_h released before the start of $\delta_{i,k}$ will either not preempt τ_i at all, or execute during a preemption before the start of $\delta_{i,k}$. An instance released after the start of $\delta_{i,k+1}$ clearly did not affect a preemption at $PP_{i,k}$. ◀

Note that an instance released after the end of $\delta_{i,k}$ cannot *cause* a preemption at $PP_{i,k}$, but it can still affect the preemption overhead if it arrives during the execution of some other task that caused the preemption.

▶ **Corollary 2.** *If task τ_h affects the preemptions at both preemption points $PP_{i,k}$ and $PP_{i,l}$ (where $k < l$) in the same instance of τ_i, then two instances of τ_h were released between the start of $\delta_{i,k}$ and the start of $\delta_{i,l+1}$.*

Proof. A single instance of τ_h cannot affect both preemptions, and the interval in which they were released is given by Lemma 1. ◀

▶ **Proposition 3.** *If $I_i^{k,l} \leq T_h$ then task τ_h cannot affect one instance of τ_i at both preemption points $PP_{i,k}$ and $PP_{i,l}$.*

Proof. Proof by contradiction: Assume that $I_i^{k,l} \leq T_h$ and that τ_h affects one instance of τ_i at both preemption points $PP_{i,k}$ and $PP_{i,l}$. Then, by Corollary 2, two instances of τ_h were released between the start of $\delta_{i,k}$ and the start of $\delta_{i,l+1}$, and thus within an interval of length $I_i^{k,l}$. This contradicts the minimum inter-arrival time T_h. ◀

Finally, per τ_i we generate a set C of constraints stating that at most one of $X_{h,k}$ and $X_{h,l}$ be true at the same time, for all cases where the inequality $I_i^{k,l} \leq T_h$ holds:

$$C = \left\{ X_{h,k} + X_{h,l} \leq 1 \;\middle|\; (1 \leq h < i) \wedge (k+1 \leq l \leq d) \wedge (I_i^{k,l} \leq T_h) \right\}.$$

Continuing the running example from Figure 2, we first compute the following $I_i^{k,l}$ values: $I_3^{1,2} = 94$, $I_3^{2,3} = 83$ and $I_3^{1,3} = 167$. In order to generate constraints, we check the proposed inequality by comparing the derived values with $T_1 = 100$ and $T_2 = 130$ and e.g, get that: $I_3^{1,2} \leq 100$, therefore, we generate the constraint $X_{1,1} + X_{1,2} \leq 1$ denoting that τ_1 cannot directly affect the preemption cost of both preemption points: $PP_{3,1}$ and $PP_{3,2}$. The complete set of constraints for τ_3 of the running example is:

$$C = \Big\{ \underset{\uparrow}{X_{1,1} + X_{1,2} \leq 1}; \quad \underset{\uparrow}{X_{1,2} + X_{1,3} \leq 1}; \quad \underset{\uparrow}{X_{2,1} + X_{2,2} \leq 1}; \quad \underset{\uparrow}{X_{2,2} + X_{2,3} \leq 1} \Big\}.$$

$$I_3^{1,2} \leq 100 \qquad I_3^{2,3} \leq 100 \qquad I_3^{1,2} \leq 130 \qquad I_3^{2,3} \leq 130$$

4.3 Goal function formulation

For the final part of the constraint problem, we define the following goal function:

$$G = \mathit{Maximize} : \sum_{PP_{i,k} \in \tau_i} \sum_{m \in UCB_{i,k}} \min\left(1, \sum \{X_{h,k} \mid \tau_h \in hp(\tau_i) \wedge m \in ECB_h\}\right) \times BRT$$

The minimum function stands for the calculation of the impact of single memory block m that may be in the ECB set of many higher priority tasks but still only contributes at most once to the preemption cost of one preemption. During the generation, *min* functions that only contain a single $X_{h,k}$ variable (or none) can be simplified.

The optimized goal function for τ_3 of the running example is:

$$
\begin{array}{lllllllll}
G = \mathit{Maximize} : BRT \times \big(& & & & & & & & \\
X_{1,1} & + & & & & & & & \leftarrow k = 1 \\
X_{1,2} & + & \min(1, X_{1,2} + X_{2,2}) & + & X_{2,2} & + & & & \leftarrow k = 2 \\
 & & X_{1,3} & + & X_{2,3} & + & X_{2,3} \big) & & \leftarrow k = 3 \\
 & & & & & & & & \\
\uparrow & & \uparrow & & \uparrow & & \uparrow & & \\
m = 1 & & m = 2 & & m = 3 & & m = 4 & &
\end{array}
$$

The m and k values indicate the different parts of the nested sum of the goal function by denoting the actual memory block m and the preemption point k that are considered by the specified element of the sum. The empty positions in the array come from the definition of the second sum over $m \in UCB_{i,k}$. For example, since the useful cache block set of the first preemption point of τ_3 is $UCB_{3,1} = \{1\}$, the first row ($k = 1$) intersects only with the $m = 1$ column. In this example, only one *min* function remained after the simplification, since memory block $m = 2$ is the only block that is in the ECB sets of both preempting tasks τ_1 and τ_2.

The final result from the solver for the running example corresponds to the preempting scenario where $PP_{3,2}$ is affected by τ_1 and $PP_{3,3}$ is affected by τ_2. Concretely, τ_1 evicts the useful cache blocks 2 and 3 from $UCB_{3,1}$, and τ_2 evicts useful cache blocks 3 and 4 from $UCB_{3,3}$. Since the block reload time is $BRT = 1$, the finally computed upper-bounded preemption cost of the task τ_3 is $\gamma_3 = 4$.

5 Experimental Results

We conducted two experiments, using the open-source Java constraint programming library Choco [9], in order to investigate the possibility of tightening the CRPD per taskset using the proposed method. Each point in the experiment represents an average of applying the algorithm to 2000 randomly generated tasksets.

In each experimental setup we generate tasksets consisting of the defined number of tasks. Each taskset is generated with the utilization level of 0.8 and the tasks' individual utilizations are generated using the U-unifast algorithm [5]. Then, the minimum inter-arrival time for each task is generated using a uniform distribution from the range $[5ms, 5s]$, thus reasonably corresponding to real systems. Next, we calculate the execution times such that $C_i^{np} = U_i \times T_i$ and assign priorities in a rate monotonic order. The number of resulting non-preemptive regions are generated from the uniform distribution in range $[1, 10]$.

Regarding the cache simulation we represent evicting and useful cache block sets using the integer values from 0 to 256, corresponding to the maximum cache set size $CS = 256$. Following the setup used by Altmeyer et al. [3], we use the following values:

- $BRT = 8\mu s$.
- ECB_i for each task is generated using the U-unifast algorithm such that the total cache utilization $CU = \sum_1^n |ECB_i|/CS$. If the utilization of the evicting set is above 1, it means that it uses all cache sets.
- UCB_i for each task is generated from the ECB_i using the reload factor RF which is uniformly generated from the range $[0, 0.3]$, thus $UCB_i = RF \times |ECB_i|$ where $|ECB_i|$ is

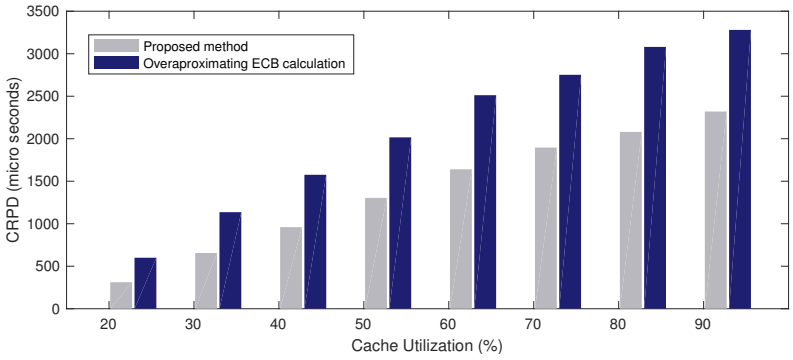


Figure 3 CRPD estimation per taskset for different levels of cache utilization, calculated as the average over the 2000 generated tasksets.

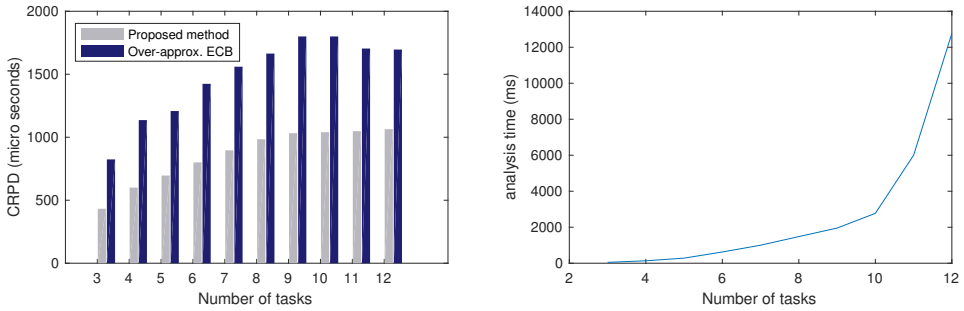


Figure 4 Left: CRPD estimation per taskset, for different taskset sizes, calculated as the average over the 2000 generated tasksets. Right: Analysis time of the proposed method per taskset, for different taskset sizes.

the number of cache sets in the ECB_i. The reload factor is used to adapt the assumed reuse factor, thus reflecting systems with a low reuse up to control-based applications with heavy reuse (up to 0.3).

- Each $UCB_{i,k}$ is generated using the uniform distribution from $[0, 100 \times |UCB_i|]$.

In the first experiment, we evaluated the CRPD estimation varying the total cache utilization from 20% to 90% (see Figure 3). Taskset utilization was fixed to 0.8 and the number of tasks was fixed to 10. We show the CRPD values computed by the proposed method, compared to the method where the worst case eviction scenario is assumed for each preemption point. As expected, the proposed method succeeds in tightening the CRPD value by identifying infeasible eviction cases. The increase of cache utilization is followed by an increase of the computed CRPD values. However, we see that the CRPD reduction ratio is decreasing by the increase of the cache utilization (when $CU = 20\%$, the reduction ratio is 49% and when $CU = 90\%$ it is less than 30%). This is the case because of the fact that with high cache utilization each higher priority task evicts a significant part of the total cache. Therefore, the benefit of identifying infeasible eviction cases is reduced since the remaining cases still evict much of the total cache in each point.

In the second experiment, we evaluated the CRPD estimation varying the taskset size from 3 to 12 (see Figure 4). Taskset utilization was fixed to 0.8 as well as the total cache utilization, $CU = 40\%$. From 3 to 9 tasks, the ratio between the CRPD estimated by the proposed method and the over-approximating ECB computation remains roughly the same

(48% for 3 tasks in a taskset, and 42% for 9 tasks). From this point, the over-approximated CRPD values drop because the evicting cache blocks of the tasks are significantly reduced as we fix the total cache utilization per taskset. Therefore, considering 12 tasks in a taskset the CRPD reduction ratio is 35%. To investigate scalability of the approach, we also measured the analysis time as the number of tasks increases. We can see that on average the method needs approximately 12 seconds for tasksets consisting of 12 tasks, and only 48 *ms* for tasksets consisting of 3 tasks. Note that the analysis time varies a lot for different tasksets, and in this experiment there are some cases for which the constraint solver needs several minutes to solve the given problem. Therefore we used a time limit of 40 seconds, and if the method failed to provide a value within this time, it instead reported the over-approximated CRPD value.

6 Conclusions

In this paper we proposed a novel method for computing the cache-related preemption delay (CRPD) in sporadic task model scheduled under the Fixed Preemption Point approach. The method identifies infeasible preemption combinations for a given preempted task based on scheduling level analysis and then calculates the maximum CRPD value considering the remaining preemption combinations and cache level analysis information. Furthermore, we defined an algorithm that iteratively estimates the upper bound on CRPD values in a taskset, using the derived upper bounds for computing the tighter CRPD estimates for lower priority tasks. In the evaluation of the proposed method, we showed that it can significantly reduce the CRPD of a preempted task and moreover all tasks in a taskset, up to 50% as shown in some cases.

For future work, we envision a preemption point selection algorithm that will exploit the proposed method for CRPD computation to achieve CRPD minimisation by the appropriate preemption point placement, benefiting from the information of the infeasible evicting scenarios defined in this paper. We also plan to investigate a way to manage the scalability limitations by restricting the algorithm to consider only a selected subset of tasks. In particular, the higher priority tasks since any tightening of their CRPD impacts the response times of all lower priority tasks, but there might be also other indicators to be included in the analysis. Another way to manage scalability limitations would be to combine the multiset-based CRPD estimation approaches with the analysis of infeasible preemption combinations in order to derive safe but to some extent pessimistic CRPD upper-bounds, compared to those computed with the constraint solving approach.

Finally, future work will also focus on developing an analogical method for tightening the CRPD bounds in fully preemptive systems. In this context, we envision a method that would first virtually divide the preempted task into subsections such that it accounts for the variability of useful cache blocks throughout the execution of the preempted task. By analysing the infeasible preemption scenarios among those subsections we would be able to furthermore tighten the CRPD bounds in such systems as well.

Acknowledgements. We are very grateful to Peter Puschner and TACLe summer school for the inspiration and provided knowledge which helped us to pursue this line of research. We would also like to thank the anonymous reviewers for their comments on an earlier draft of this paper and the suggestions for the continuation of the work.

References

1. Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.
2. Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. *ACM Sigplan Notices*, 45(4):153–162, 2010.
3. Sebastian Altmeyer, Douma Roeland, Will Lunniss, and Robert I. Davis. Evaluation of cache partitioning for hard real-time systems. In *Proceedings Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26, 2014. doi:10.1109/ECRTS.2014.11.
4. Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 217–227. IEEE, 2011.
5. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
6. José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings.*, pages 204–212. IEEE, 1996.
7. Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. A survey. *industrial Informatics, IEEE Transactions on*, 9(1):3–15, 2013.
8. John Cavicchio, Corey Tessler, and Nathan Fisher. Minimizing cache overhead via loaded cache blocks and preemption placement. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 163–173. IEEE, 2015.
9. CHOCO: Open Source Java Library for Constraint Programming. http://www.choco-solver.org/. Accessed: 2017-04-13.
10. Chang-Gun Lee, Joosun Han, Yang-Min Seo, Sang Luyl Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sam Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.
11. Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in cost-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231. IEEE, 2008.
12. Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 177–188. IEEE, 2014.
13. Stefan M Petters and Georg Färber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems, London, UK*, 2001.
14. Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):27, 2010.
15. Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48. IEEE, 2005.

Early WCET Prediction Using Machine Learning

Armelle Bonenfant[1], Denis Claraz[2], Marianne de Michiel[3], and Pascal Sotin[4]

1 **University of Toulouse, IRIT, Toulouse, France**
`bonenfant@irit.fr`
2 **Continental Automotive, Toulouse, France**
`denis.claraz@continental-corporation.com`
3 **University of Toulouse, IRIT, Toulouse, France**
`michiel@irit.fr`
4 **University of Toulouse, IRIT, Toulouse, France**
`sotin@irit.fr`

Abstract

For delivering a precise Worst Case Execution Time (WCET), the WCET static analysers need the executable program and the target architecture. However, a prediction – even coarse – of the future WCET would be helpful at design stages where only the source code is available. We investigate the possibility of creating predictors of the WCET based on the C source code using machine-learning (work in progress). If successful, our proposal would offer to the designer precious information on the WCET of a piece of code at the early stages of the development process.

1998 ACM Subject Classification D.4.7 Real-Time Systems and Embedded Systems

Keywords and phrases Early WCET, Machine Learning, Static Analysis, C Language

Digital Object Identifier 10.4230/OASIcs.WCET.2017.5

1 Introduction

Context and Motivation

The Worst Case Execution Time (WCET) static analysers operate – for most of them [10, 7, 3, 12] – on the binary of the program under analysis, taking into account the target architecture. With these elements the WCET estimate can be proven safe and might hopefully be precise. Unfortunately these requirements make that the WCET estimate is available late in the development process when many choices linked to the WCET have already been done.

▶ **Situation 1.** *At Continental Automotive, any new SW project version is based on already existing components on the shelf, as well as newly developed components. The existing components usually have associated WCET (obtained by measurement) for some architectures. For the components that have no timing attached or only timings for too distinct architectures, a tool operating on source code and providing a prediction of the future WCET would be of great help. In particular in multi-core context, an early estimation of the runtime would help the integrator to choose the right core for integration.*

Work In Progress

We are currently investigating the possibility of applying machine learning to obtain source-code-based WCET predictors (specific to the compilation chain and architecture on which they were learnt). In this article:

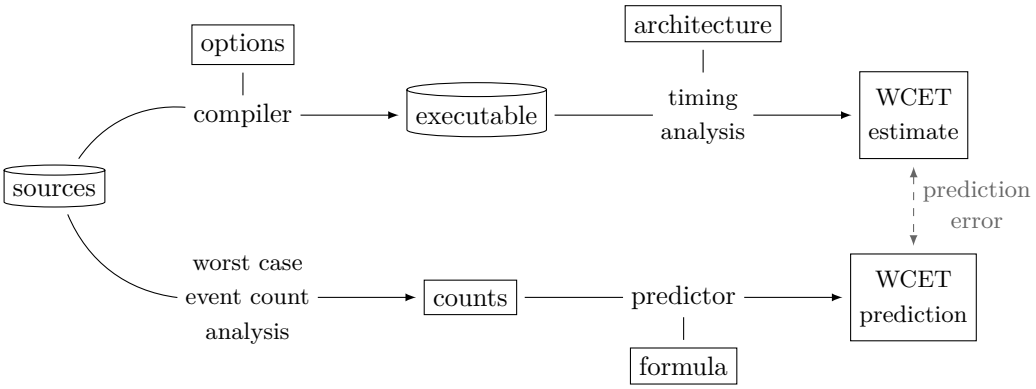


Figure 1 Two ways to approach the WCET.

- We introduce a framework for creating the so-called predictors (Section 2). This framework rely on *static analysis* of the source code, on *machine learning* and on a set of programs which final WCET estimate is known or can be computed.
- We present a novel source code static analysis that counts certain events occurring in the worst case execution (Section 3).
- We discuss the application of machine learning in our setting (Section 4). In particular, we list several questions that have not received satisfying answers yet.

2 Overview

In some industrial settings, as illustrated in Situation 1, the size of the project (e.g. 1.5 M effective lines of code) or the complexity of the compilation chain makes it time consuming to create the program binary and thus to obtain the WCET. In addition, the company organisation might imply that distinct persons from distinct departments provide the source code, integrate and build the binary. Due to the size of the projects, and the short development times, a parallelization of the developments is mandatory, and therefore iterative integration cycles are done. When doing an integration step, having an idea of the future WCET of a given component (newly developed, or old but not measured nor analysed) is important since it might influence the selection of the core, the position in the task, or in worst case, the whole project architecture. Early availability of the information is here more important than accuracy of the result, the effective real time behaviour being tested extensively or statically analysed in a further phase. This need of an early WCET is also mentioned by [11, 6, 1].

Our current work pursues the idea that it might be possible to get a satisfying prediction of the future WCET by applying a formula on some characteristics of the source code (Figure 1). This formula has to be learnt from a set of programs which WCET estimate is already known (Figure 2).

In Figure 1, we depict two ways of getting an idea of the WCET.

- The top line is the usual WCET derivation, we call its result *WCET estimate*. This estimate is by construction an over-approximation of the WCET.
- The bottom line is our proposal. We call its result a *WCET prediction*. This prediction cannot be safely used as WCET estimate but should give an relatively close approximation of the top line WCET estimate if we can find a satisfying formula.

A similar pattern is followed by [6, 1] but relies on measurements (related work is detailed in Section 5).

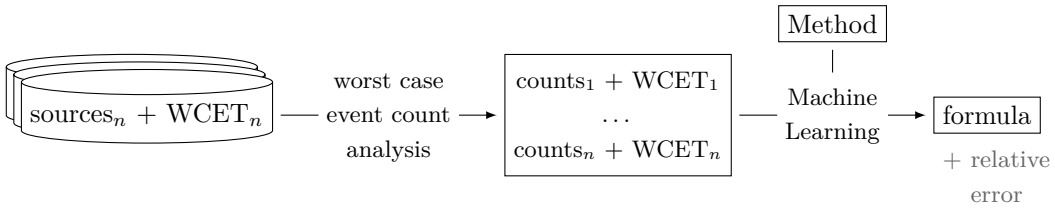


Figure 2 Learning WCET source code predictors.

Discovering a formula linking precisely enough the WCET to the source code for a given compilation chain and architecture is the main goal of our work. In order to achieve this goal we put our faith in machine learning.

First of all, we note that machine learning cannot be applied straightforward on programs. The best it can deliver us is to link the WCET estimate with discrete or continuous attributes of the program source code. We identified two classes of attributes that can be fed to the machine learning:

1. *Count* attributes. For example the maximal number of arithmetic operations, function calls or global variable read accesses. These attributes give a concrete idea of the program duration and could be assembled by the learning tool into a nice formula.
2. *Style* attributes. For example the number of lines of code, the maximal loop nesting, whether the program was generated or hand-written. These attributes are by themselves insufficient to get an idea of the program duration but they can help the learning tool to define classes of programs.

We defined a static source code analysis that we call *worst case event count analysis*. It extracts count attributes (Item 1) from the source code. This analysis provides more than source-code-based WCET analysis [9] since it delivers counts for several categories of events and not a single already-aggregated WCET estimate. In our process, aggregation is left to machine learning. Details are given in Section 3.

The worst case event count analysis is performed in two situations:

- Before applying the formula on a given source code (bottom line of Figure 1),
- Before feeding a set of programs to the machine learning process (first step of Figure 2).

Figure 2 depicts the learning process. This process starts with a set of programs for which the WCET estimates are known. The only assumption that we make on these WCET estimates is that they intend to reflect the WCET of the source code compiled for a given architecture. The worst case event counts are then extracted for these programs and paired with the WCET estimates. The spreadsheet obtained is then fed to the machine learning tool that will output a formula predicting the WCET estimate in function of the event counts. Depending on the method used by the tool the formula can range from a linear combination of the events to a neural network.

In Section 4 we discuss the problems we face in that learning process.

3 Count Attributes Extraction

In this section, we present how we extract worst case event counts from the C source code. Our proposal is linked with source code timing scheme [9] but the data manipulated are more complex (count by categories and sets of such metrics).

(a) X is smaller than Y in each category

(b) X and Y are not ordered as in Figure 3a but an optimistic Δ_Y cost more than a pessimistic Δ_X

Figure 3 Two reasons for ignoring metric X in front of metric Y.

3.1 Analysis Principles

Our analysis is an evaluation of the source code syntax, aware of the loop bounds.

Analysis Domain

We consider an `if-then-else` statement where the `then` and `else` branch evaluations gave respectively the metrics T and E such that:

$$T = [\mathit{Arith} \mapsto 10; \mathit{Load} \mapsto 3], \qquad E = [\mathit{Arith} \mapsto 2, \mathit{Load} \mapsto 5]\,.$$

Where *Arith* and *Load* respectively stand for *performing a basic arithmetic operation* and *read access to a variable*. What should then be the evaluations of the `if-then-else` statement? An imprecise but "sound" solution would be to use the smallest metric greater than T and E, also known as the least upper bound $T \sqcup E$ of T and E:

$$\begin{aligned} T \sqcup E &= [\mathit{Arith} \mapsto \texttt{max}(10, 2); \mathit{Load} \mapsto \texttt{max}(3, 5)] \\ &= [\mathit{Arith} \mapsto 10; \mathit{Load} \mapsto 5]\,. \end{aligned}$$

This option is appealing because it seems to mimic the over-approximation that will be performed by the timing analysis. This is not the case. When the binary WCET analysis has to select the `then` or the `else` branch it reasons on numbers of cycles that are totally ordered. The solution it retains might really be the worst case. In our case, the metric $T \sqcup E$ is wrong: no "execution" can reach these figures. Using the average or the minimum instead of the maximum would not deliver a more satisfying result.

The metrics are structured as a partially ordered set (poset) isomorphic to $\mathbb{N}^{\mathbf{c}}$ where $\mathbf{c}$ is the set of event categories. A precise solution is to to keep track of *sets of maximal metrics*. For the `if-then-else` statement above the evaluation would be the set $\{T, E\}$ since none of them is greater than the other. Potentially an evaluation could contain as many elements as the number of permutations on $\mathbf{c}$ ie. $|\mathbf{c}|!$. In practice the size of the set does not grow that fast and we use some architectural knowledge in order to state for example that:

$$[\mathit{Load} \mapsto 5; \mathit{Store} \mapsto 2] \prec [\mathit{Load} \mapsto 40; \mathit{Store} \mapsto 1]\,.$$

The argument that sustains this ordering is the fact that one extra store cannot be more expensive than 35 extra loads. More formally, each category receives a pessimistic and an optimistic evaluation in terms of cycles. With these ranges we define the two functions $\mathit{eval}_{\text{pess}}$ and $\mathit{eval}_{\text{opt}}$ that assign a numerical value to a metric. We then redefine the order on the metrics as follows:

$$X \prec Y \iff \mathit{eval}_{\text{opt}}(X - Z) < \mathit{eval}_{\text{pess}}(Y - Z) \qquad \text{with } Z = X \sqcap Y\,.$$

Table 1 Categories used by our worst case event count analysis.

Family	Category	Optimistic	Pessimistic
Operations	Simple	0.5	1
	Multiplication	1	5
	Division	1	10
Control	Unconditional branch	0.5	1
	Conditional branch	1	1
	Computed branch	1	4
	Call	0.5	10
Memory	Address setting	0.5	1
	Load	2	20
	Store	2	20

where the metric $X \sqcap Y$ is the greater lower bound of X and Y (it is computed by keeping the minimum for each category) and the metric $X - Z$ is the difference of X and Z, category by category. This ordering on metrics encompasses the point-wise ordering. Figure 3 can give the visual intuition of this ordering.

Eventually, the set of maximal metrics computed for the whole program needs to be brought back to a single metric in order to be fed to the learning process. At that stage there is often few elements R_i in the set because when numbers of events get large, the ordering eliminates many metrics that cannot be the worst-case path. We approximate to $R_1 \sqcup \ldots \sqcup R_n$.

Analysis Rules

The sets of maximal metrics ($\mathcal{A}$, $\mathcal{B}$) are combined according to these rules:

$\mathcal{A}$ and $\mathcal{B}$ are in sequence	$\uparrow \{X + Y \mid X \in \mathcal{A}, Y \in \mathcal{B}\}$,
$\mathcal{A}$ and $\mathcal{B}$ are alternatives	$\uparrow (\mathcal{A} \cup \mathcal{B})$,
$\mathcal{A}$ is repeated n times	$\{n \times X \mid X \in \mathcal{A}\}$,

where the metric $X + Y$ is the sum of X and Y, category by category, the metric $n \times X$ is the multiplication of each category of X by n and the set $\uparrow \mathcal{S}$ is the set of metrics $\mathcal{S}$ where non-maximal elements have been removed.

3.2 Implementation

We implemented the analysis presented in Section 3.1 on top of the oRange static analysis tool for C [8]. Table 1 shows the considered categories and associated optimistic and pessimistic evaluation for events of these categories (in cycles).

4 Machine Learning

In this section, we present our machine learning environment (Section 4.1), discuss the difficulties we face setting up the framework of Figure 2 (Section 4.2) and present our plans for the near future (Section 4.3).

4.1 Machine Learning Methods

We will use WEKA [5], a machine learning software. It is a collection of state-of-the-art visualization tools and algorithms for data analysis and predictive modelling. Among other techniques, it supports data preprocessing, clustering, classification and regression.

Once data has been collected on a large set of programs, the software will provide us classifications, and ways to visualize our data. It allows the systematic comparison of the predictive performance of WEKA's machine learning algorithms on a collection of datasets.

WEKA can be set up to compute the error while learning: the data is divided in 10 packs; in turn, 9 are used to learn and remaining one is used to compute the error.

The validity of the learnt formula depends on the quantity and the quality of the data.

4.2 Learning WCET Predictors

Program Set

In order to have a precise learning, the program set should be large and representative – preferably thousands of elements. Finding such a set of programs is definitely an issue:

- Benchmarks, like the TACLeBench [4], seldom contain more than a hundred programs. These programs are representative but heterogeneous (size, style, purpose).
- Program generators, like CSMITH [13], offer as many programs as needed. These programs might be far from representative. In addition, their lack of meaning makes that clever compiler may optimize them in unpredictable proportions (dead code elimination, pre-computation) thus breaking any correlation between the source and the binary.
- Industrial program base, like the one mentioned in Situation 1, may contain hundred of software components. These program are representative and homogeneous.

The set of programs used should also be correlated with the categories of event considered. Events that are absent or simply invariant in the program set might be integrated to the formula in an absurd way.

Choice of the Learning Method

The WEKA machine learning application offers a *huge* number of learning methods, some of them presenting tricky parameters. The exploration of this possibilities is a non-trivial dimension of our problem.

Validation of the Predictor

As mentioned in Section 4.1, the machine learning algorithms already offer a notion of relative error. When considering large programs, we would consider as successful a relative error of $\pm 20\%$.

If the chosen method delivers us a weight for each event category we can validate that:

- The weights are within the optimistic/pessimistic ranges used in Section 3.
- The weights are sensible values for a WCET expert.

Relevant Event Categories

The categories listed in Table 1 where chosen because they seem relevant from a WCET point-of-view. For example, multiplication and division have been set aside from the other operations because their compilation can range from a shift to an emulation function.

Some of these categories can be shown to be useless by our future work while some might need to be refined. If our proposal is successful, the final categories could be considered as an interesting production.

Feasibility

Eventually, we need to consider this hypothesis:

▶ **Hypothesis 2.** *The formula we are looking for does not exists.*

For a given compilation process and micro-architecture Hypothesis 2 means that it is impossible to correlate any source code characteristics to the WCET of the binary. This hypothesis is not absurd since the impact of the compilation and the hardware on the WCET can be huge and far from regular. This hypothesis cannot be proven because failing to learn can be the consequence of either nothing to learn or bad learning method.

If Hypothesis 2 holds, then we propose two ways to fight against it:

- Make a step toward the binary by performing the front-end compilation passes, then applying the worst case event count on the intermediate representation. Hopefully these passes will be common to most compilation chains.
- Restrict our claim to a family of programs.

4.3 Roadmap

A modified version of the static analyser oRange [8] provides us the worst case event count analysis. The WCET analysis framework Otawa [2] will be used to compute the WCETs of the program base. The machine learning will be provided by the Weka platform.

We have proceeded to a first experiment in order to get a proof of concept. We did the experiment on random generated programs. The learnt formulae were too complex and the imprecisions were too wide. We believe that this failure is due to the use of programs having WCET of different magnitudes. We tried to learn from programs being too different.

Thus, for this work, our roadmap is as follow:

- Getting a random program generator in order to calibrate the programs. We want to get a classification prior to submit to Weka: program having similar size, similar structure (loops/no loop, conditions/no condition, nested/non nested loops, number of event...). At first, we used CSmith [13] but we need more calibration possibilities.
- For each category of programs, getting a formula
- Combining the class of programs and getting a new valid formula
- Classifying the programs of a benchmark (or use case) and running the formula.

5 Related Work

In [6, 1], Gustafsson et al. address the same problem with a different tool. From a set of measurement on several programs, they infer a timing model. Once learnt, this timing model can be applied to the measurements of a new piece of code. The technique used to learn the model is a variation of a linear regression adapted to the fact that only some of the measurements might reflect worst case behaviours. This technical point prevents the exploration of the learning techniques we can afford since it would lead to learning an average behaviour. A important distinction between this work and our proposal is that our proposal requires no access to the effective hardware but an already set up static analysis

tool (e.g. OTAWA and the hardware model) while their work requires no pre-existing model but an access to the execution platform.

As already mentioned in Section 2 and in the header of Section 3 our proposal offer a feature similar to [9] with the difference that we transfer the task of aggregating cleverly the software events to a machine learning tool. Our proposal is thus more complex (need worst case event count analysis) but we benefit from machine learning and we can (try to) learn any architecture.

6 Conclusion

Early WCET evaluation is a recurrent request in the domain. Our approach face two main issues: finding relevant attributes and applying relevant machine learning analysis.

Our expertise on classical WCET evaluation by static analysis is solid. We think we are able to identify and compute qualitative attributes for C programs.

The second issue will obviously require machine learning expertise in addition to time analysis expertise. Once the programs are classified, we are confident that we will be able to obtain a calculus for early WCET. Applying this approach to general programs is more ambitious, our study will tell the conditions of applications.

Acknowledgements. We wish to thank Mathieu Serrurier for patiently sharing with us bits of its expertise in machine learning.

References

1 Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, 2016. doi:10.1007/s11241-016-9250-7.

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

3 Franck Cassez and Jean-Luc Béchennec. Timing analysis of binary programs with UPPAAL. In Josep Carmona, Mihai T. Lazarescu, and Marta Pietkiewicz-Koutny, editors, *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 41–50. IEEE Computer Society, 2013. doi:10.1109/ACSD.2013.7.

4 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 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 fuer Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

5 Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H. Witten, and Len Trigg. Weka-a machine learning workbench for data mining. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook, 2nd ed.*, pages 1269–1277. Springer, 2010. doi:10.1007/978-0-387-09823-4_66.

6 Jan Gustafsson, Peter Altenbernd, Andreas Ermedahl, and Björn Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In Sunggu Lee and Priya Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop, SEUS 2009, Newport Beach, CA, USA, November 16-18, 2009, Proceedings*, volume 5860 of *Lecture Notes in Computer Science*, pages 308–319. Springer, 2009. doi:10.1007/978-3-642-10265-3_28.

7 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In Richard Gerber and Thomas J. Marlowe, editors, *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995). La Jolla, California, June 21-22, 1995*, pages 88–98. ACM, 1995. doi:10.1145/216636.216666.

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

9 Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991. doi:10.1109/2.76286.

10 Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.

11 David Trilla, Javier Jalle, Mikel Fernández, Jaume Abella, and Francisco J. Cazorla. Improving early design stage timing modeling in multicore based real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 305–316. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461338.

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

13 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.

Worst-Case Execution Time Analysis of Predicated Architectures*

Florian Brandner[1] and Amine Naji[2]

1 **LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France**
florian.brandner@telecom-paristech.fr
2 **U2IS, ENSTA ParisTech, Université Paris-Saclay, Palaiseau, France**
amine.naji@ensta-paristech.fr

Abstract

The time-predictable design of computer architectures for the use in (hard) real-time systems is becoming more and more important, due to the increasing complexity of modern computer architectures. The design of predictable processor pipelines recently received considerable attention. The goal here is to find a trade-off between predictability and computing power.

Branches and jumps are particularly problematic for high-performance processors. For one, branches are executed late in the pipeline. This either leads to high branch penalties (flushing) or complex software/hardware techniques (branch predictors). Another side-effect of branches is that they make it difficult to exploit instruction-level parallelism due to control dependencies.

Predicated computer architectures allow to attach a *predicate* to the instructions in a program. An instruction is then only executed when the predicate evaluates to true and otherwise behaves like a simple `nop` instruction. Predicates can thus be used to convert control dependencies into data dependencies, which helps to address both of the aforementioned problems.

A downside of predicated instructions is the precise worst-case execution time (WCET) analysis of programs making use of them. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache and thus need to be considered by the cache analysis. Predication potentially has an impact on all analysis phases of a WCET analysis tool. We thus explore a preprocessing step that explicitly unfolds the control-flow graph, which allows us to apply standard analyses that are themselves not aware of predication.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems

Keywords and phrases Predication, Worst-Case Execution Time Analysis, Real-Time Systems

Digital Object Identifier 10.4230/OASIcs.WCET.2017.6

1 Introduction

Embedded real-time systems, as most computer systems, faced a steady increase in requirements [4] during more than three decades. An increase in requirements frequently also translates into an increased performance need. Simple and predictable micro-controllers cannot satisfy these needs in many cases. To address this issue new architecture designs have recently been explored that promise a high degree of (time-)predictability while offering an acceptable performance level [20, 16, 23, 18]. Due to its high-latency, designs focusing on the memory hierarchy have been explored extensively [17, 13, 19, 2]. Also the design

* This work was supported by a grant (2014-0741D) from Digiteo France: "Profiling Metrics and Techniques for the Optimization of Real-Time Programs" (PM-TOP).

of processor pipelines has recently received considerable attention [3, 20, 16, 24]. The goal behind all this work is to find a trade-off between time-predictability and computing power.

The handling of branches and jumps is particularly problematic for the design of time-predictable high-performance processors. The new value of the program counter (`PC`) can usually only be computed late in the processor pipeline.[1] As the branch advances through the pipeline the processor is unable to tell which instructions need to be fetched next. A simple solution is to simply fetch the instructions immediately after the branch. These instructions are typically *flushed* from the pipeline, i.e., discarded, once the actual value of the `PC` is available. This solution induces a considerable branch penalty. *Branch prediction* techniques may be used to guess the branch direction and/or address earlier. Static techniques rely on the compiler to provide good predictions, while dynamic predictors require additional hardware. Both approaches allow to reduce the branch penalty. However, the state of hardware-based branch predictors needs to be taken into consideration during *Worst-Case Execution Time* (WCET) analysis. Conflicts between branches of the program itself and interference from other tasks further complicate the analysis. An alternative solution is to let the instructions following a branch execute. The instructions in these *branch delay slots* might then perform useful work – if the compiler is able to rearrange the instructions accordingly. This approach has drawbacks, despite being predictable: (a) unused branch delay slots have to be filled with `nop` instructions and thus increase code size and (b) tools, including the compiler and WCET analyzer, have to be aware of the branch delay slots.

Branches, furthermore, introduce control dependencies, i.e., instructions *after* a branch can only be executed safely when the branch direction/address has been determined. This even applies when branch prediction is used: the processor may only execute instructions *speculatively* as long as the instructions do not cause any side-effects.[2] Inversely, instructions *before* the branch can usually not be moved to locations after the branch. Branches consequently make it more difficult to exploit *instruction-level parallelism* [7].

One solution to these issues is *predication*, where an additional predicate is attached to instructions. The predicate is evaluated at runtime and allows to conditionally nullify the effect of the instruction, i.e., the instruction is discarded and behaves like a simple `nop` instruction when the predicate evaluates to false. The aforementioned problems can be addressed using predication by moving instructions past branches. The control dependence, with regard to the original branch instruction, is then effectively transformed into a data dependence on the predicate. This, for instance, simplifies the compiler's task to fill branch delay slots or to exploit instruction-level parallelism. In some cases branches can even be eliminated completely. This is particularly advantageous for short code sequences, where the branch penalty often outweighs the cost of executing a few predicated instructions. Some real-time systems actually take advantage of the possibility to entirely eliminate all conditional branches. This is known as the single-path programming paradigm [15].

Despite these advantages, predicated instructions can be problematic during WCET analysis. Side-effects of predicated instructions need to be analyzed, which depend on the runtime value of the instruction's predicate. This may have an impact on many analysis steps, including value range analysis, loop bounds analysis, infeasible path analysis, but also the cache and pipeline analyses. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache, depending on the predicate. The simplest solution

[1] After reading register values at the level where regular arithmetic operations are often handled.
[2] Visible according to the processor's programming conventions, which usually does not include hidden states (caches, branch predictors, ...).

for the analysis would be to ignore predicates and conservatively consider the effect of both cases. This may result in very conservative results, since the implicit information available in the program's original control flow before the elimination of branches is entirely lost. The analysis could also be extended to be aware of predicates. Note, however, that this may require changes to virtually all analysis steps in a WCET analyzer and may thus require a considerable engineering effort. We thus explore a much simpler solution that consists in recovering the (hidden) control flow. Instructions that *define* (set) a predicate are handled similar to branch instructions and lead to a control-flow split. The succeeding instructions are then *duplicated*, once assuming that the predicate evaluates to `true` and once assuming that the predicate value is `false`. Subsequent instructions that are predicated with that predicate are now trivial to handle: an instruction either always corresponds to a `nop` or always corresponds to the regular unpredicated instruction.

The remainder of this paper is structured as follows. We will first give some background regarding the Patmos platform on which our work builds, covering the architecture design as well as the tool suite (compiler, analyzer). Section 4 then describes a simple algorithm that allows us to recover the hidden control flow of predicated code. We then present the results from preliminary experiments in Section 5. Related work, concerning predication in real-time systems and related analysis techniques, is finally discussed in Section 6 before concluding.

2 Background

Patmos Architecture

The Patmos architecture [20] is intended as a test bed to evaluate and design new time-predictable computer architecture concepts, covering cache designs [19, 2], on-chip networks [12], as well as instruction set architecture design. Among many other features, Patmos supports predicated execution. An additional predicate operand is attached to each instruction, which allows to refer to one out of the 8 predicate registers (`p0` through `p7`). The predicate operand, in addition, allows to invert the predicate value (e.g., `!p0`). Consequently, 4 bits of the instruction encoding are reserved for the predicate operand (3 bits for the predicate register, 1 bit for negation). The predicate registers themselves consist of a single bit each. Predicate register `p0` always evaluates to `true` and cannot be overwritten.

Predicate registers can be defined using dedicated comparison instructions (e.g., `cmpeq`), which allow to compare 32-bit integer values from general-purpose register operands and immediate values. These instructions allow to specify a destination predicate, which sets the predicate register accordingly. In addition, basic logical operations can be performed on two predicate register operands using dedicated logical-predicate instructions (e.g., `por`). The result of the operation is again written into a predicate register. Note that these instructions can be predicated themselves, which facilitates the handling of nested `if` statements.

Patmos is a *Very Long Instruction Word* (VLIW) architecture that can issue and execute multiple instructions in parallel in a five-stage, in-order pipeline: fetch (`FE`), decode (`DEC`), execute (`EX`), memory access (`MEM`), and register writeback (`WB`). Instructions are fetched from a special instruction cache, the so-called method cache [2], which guarantees that an instruction fetch always hits in the cache. Method cache misses may only occur in the `EX` stage during the execution of dedicated branch instructions (e.g., `brcf`) or function calls (e.g., `call`, `ret`). Several variations of branch and call instructions exist, having (a) no branch delay slots (e.g., `callnd`, `brcfnd`), (b) 2 branch delay slots (e.g., `br`), and (c) 3 branch delay slots (e.g., `call`, `brcf`). Similarly, misses in the data cache may only occur in the `MEM` stage. The `FE` and `DEC` pipeline stages are thus free from undesirable side-effects (apart from updating

the PC), while the EX, MEM, and WB stages may cause side-effects on the processor's registers or caches. Predicate registers are read and written in the execute stage (EX). The processor thus is able to detect whether a predicated instruction needs to be nullified before any undesirable side-effects may become visible.

Compiler

The Patmos toolchain is based on the LLVM compiler framework,[3] which compiles all source files to an internal bitcode representation. In our case, this also applies to user- and system libraries, which are also linked together at the bitcode level. The final machine-code generation is postponed to the very end of the compilation process when *all* bitcode is available [1]. The code generator thus has a complete view of the entire program, which can be optimized and analyzed before the final executable file is generated.

The LLVM code generator is able to produce predicated code at several stages. Firstly, the instruction selector is able to recognize simple *select statements* that are directly compiled to conditional moves. A generic if-conversion optimization is also available, which allows to eliminate conditional branches of complex control-flow and produce predicated code. Finally, patmos-specific code transformations are available to generate single-path programs [8].

The *Application Binary Interface* (ABI) defined by Patmos states that all predicate registers are callee saved. This means that, during a function call, the called function needs to save and restore any predicate register that it modifies. Predicate registers cannot be used to pass arguments to other functions. Predicates are, in terms of the ABI, strictly function local. Consequently, predicates can be freely used across function calls, while called functions are independent from predicates computed before entering the function. It is still possible that call instructions themselves are predicated, i.e., the function is called conditionally.

WCET Analyses

As mentioned above, all code of the final program is available within the compiler. This enables us to immediately perform machine-code-level analyses covering the entire program (and all system libraries) before the final code emission. Several such analyses [19] are available *in the compiler* whose results can be exported to other WCET analyzers or used within the compiler itself to perform further analyses or optimizations [10]. The internal analyses either operate directly on the intermediate representation of the LLVM code generator or an enriched inter-procedural representation that allows to easily attach various analysis results to the code generator's intermediate representation in a context-dependent manner.

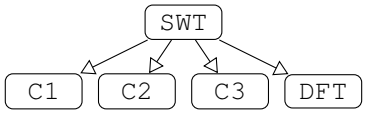
3 Motivating Example

Before giving a formal description of our proposed approach, we will give a simple motivating example. Figure 1 illustrates the implementation of a `switch` statement using a jump table and predication. The original C code is shown in Subfigure 1a and the resulting *Control-Flow Graph* (CFG) from LLVM in Subfigure 1b. Basic blocks C1 through C3 represent the three `case` statements, while basic block DFT corresponds to the `default` statement. The code of the `switch` statement itself can be found in the SWT block, whose machine code is shown in Subfigure 1c.

[3] http://www.llvm.org

```
switch(x) {
  case  1: ... break;
  case  2: ... break;
  case  3: ... break;
  default: ... break;
}
```

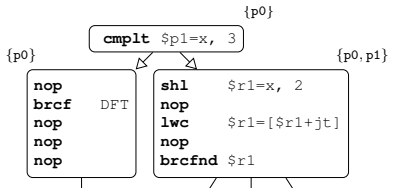
(a) Initial C code



(b) Original control-flow graph

```
        cmplt  $p1=x, 3
( $p1)  shl    $r1=x, 2
(!$p1)  brcf   DFT
( $p1)  lwc    $r1=[$r1+jt]
        nop
( $p1)  brcfnd $r1
```

(c) Code of basic block `SWT`



(d) Unfolded control-flow graph

Figure 1 Implementation of a simple `switch` statement using a jump table, the corresponding control-flow graph, the predicated machine code of basic block `SWT`, and the unfolded control flow.

The machine code uses a jump table (`jt`) that is implemented as an array holding the addresses of basic blocks `C1` through `C3`. After verifying that the value of variable `x` is within the array bounds (`cmplt`), the address of the destination block is loaded (`lwc`) and control is transferred (after a 1 cycle load delay slot) via an indirect branch (`brcfnd`). If `x`'s value exceeds the array bounds, a conditional branch (`brcf`) immediately transfers control to basic block `DFT`. Note that this branch instruction has 3 branch delay slots and that one of these slots even contains another branch instruction.

The predicated code may pose several challenges in a WCET analyzer. One particular challenge is the reconstruction of the program's CFG from the binary machine code, which usually represents the input to most WCET analysis tools. The compiler placed a branch instruction in one of the branch delay slots of another branch. In this example it is trivial to detect that the predicates of the respective branches are disjoint. However, different predicates might be used, which makes it difficult to reconstruct the actual control flow from such code. In our case this is not necessary, since the analysis is part of the compiler and thus has direct access to its intermediate representation.[4]

Another challenge, as noted before, are potential side-effects on caches or registers caused by predicated instructions. This issue is resolved by unfolding the hidden control flow from the predicated code – as depicted by Subfigure 1d. Each time when a predicate register is defined (`cmplt`) a control-flow split is performed at the level of the control-flow graph. The instruction defining the predicate is then treated in a similar way as conditional branches and subsequent code is duplicated considering both of the potential predicate values (`true` or `false`). The basic block on the left side of the subfigure here corresponds to an execution where predicate register `p1` evaluates to `false`, while the basic block on the right is executed only when `p1` evaluates to `true`. In fact, each basic block is associated with a set of predicates that are known to be `true` when entering the basic block (indicated in the top corner of each block). Inversely, predicates that do not appear in this set are known to be `false`.[5] Note that the predicates in the code are no longer needed. Predicated instructions are either duplicated *unconditionally* or are otherwise replaced by an explicit `nop`.

[4] Note that this also solves many unrelated issues during the control-flow reconstruction from binary code such as computed branch targets, function pointers, et cetera.

[5] This is safe, since LLVM inserts pseudo definitions on all program paths where a register is not defined.

In the unfolded control-flow graph it is now much easier to analyze the instructions' side-effects. The load (`lwc`) from the jump table, for instance, is only executed when the variable `x` is known to be less than 3 (`cmplt`). This means that any side-effects of this instruction on the data cache are only visible in basic blocks `C1` through `C3`, but not in basic block `DFT`. Another implicit side-effect concerns the value of variable `x` after the comparison. Due to the control-flow split at the `cmplt` instruction, it is very easy for a value range analysis to show that the value of `x` has to be larger then 3 when reaching basic block `DFT`. Delayed branches and potential redefinitions of register operands make this much more challenging in the original CFG. The algorithm to construct such an unfolded CFG, while considering predication and delayed branches, is discussed in the next section.

4 Control-Flow Unfolding

Due to space considerations, Algorithm 1 only shows a simplified version of our approach. The presented algorithm assumes a single issue architecture, which avoids the need to handle several parallel uses and (re-)definitions of predicates, parallel branches and predicate operations, et cetera. The algorithm also assumes that the predicates of branches that appear in branch delay slots are disjoint, i.e., only a single branch is known to be taken at any moment at runtime. Lastly, the presented approach only operates on the CFG of a single function. Extensions to the algorithm, included in the actual implementation, which allow us to handle these cases are briefly highlighted later. Finally, the algorithm invokes several helper functions whose code is not shown. We will briefly define these functions in an informal way before discussing the algorithm in detail.

4.1 Helper Functions

Several helper functions are needed in order to operate on individual instructions in LLVM's intermediate representation. The function Next allows to obtain the instruction immediately following an instruction i in its parent basic block, PKill returns the set of predicate registers whose live ranges end after instruction i, while the functions PDef and IsPredDef allow to obtain/test whether an instruction defines a predicate register. The function IsNop is used to test whether an instruction i is nullified given the current set of predicates P.

Several helper functions are related to branches, allowing to test for branch instructions (IsBranch), obtain the number of the branche's delay slots (BranchDelay), and obtain the successor basic blocks to which control may be transferred by a branch (BranchTargets). FallThroughTarget is used to obtain the fall-trough target basic block of the last instruction of a basic block, i.e., control is transferred to another basic block without an explicit jump or branch instruction.

Finally, three functions are related to the construction of the enriched intermediate representation of our analysis tool. GetCFNode allows to obtain the control-flow node associated with a start instruction f, the remaining number of delay slots d, and a set of predicates P – if such a control-flow node was created before. Nodes are created using the function MakeCFNode, which *duplicates* all instructions between the instruction f and e provided as arguments. The new node is also associated with d, the number of branch delay slots remaining, and P, the set of predicates. Finally, MakeCFEdge creates control-flow edges between two control-flow nodes provided as arguments.

Algorithm 1 Simplified algorithm to recover the hidden control flow from predicated code by code duplication on a single-issue architecture.

```
 1: function UnFold(MachInstr f, MachInstr l, Pred d, BasicBlockSet T, PredSet P)
 2:     if n =GetCFNode(f, d, P) then return n          ▷ Check if control-flow node exists
 3:     PredSet L = P; Pred pd = p0; MachInstr e = f          ▷ Initialize variables
 4:     for each instruction i between l and f do
 5:         L = L \ PKill(i)                                  ▷ Remove dead predicates
 6:         e = i                                        ▷ Track end of control-flow node
 7:         if ¬IsNop(i, P) then                          ▷ Skip nop instructions
 8:             if IsPredDef(i) then                        ▷ Predicate definition
 9:                 pd = PDef(i)                          ▷ Track defined predicate
10:                 break                          ▷ Immediately split control flow
11:             else if IsBranch(i) then                      ▷ Branch instruction
12:                 d = BranchDelay(i)                  ▷ Track branch delay slots
13:                 T = BranchTargets(i)                ▷ Track branch target(s)
14:         if d = 0 then break                ▷ Split control flow after branch delay
15:         d = d − 1                          ▷ Update remaining branch delay slots
16:     CFNode n = MakeCFNode(f, e, d, P)              ▷ Create a new control-flow node
17:     if e = l ∧ T = ∅ then                                  ▷ Handle fall-through
18:         T = FallThroughTarget(l)
19:     else if d ≠ 0 ∧ pd ≠ p0 then          ▷ Handle split due to predicate definition
20:         for each P′ ∈ {L ∪ pd, (L \ pd) ∪ {p0}} do      ▷ Compute successor predicates
21:             CFNode n′ = UnFold(Next(e), l, d, T, P′)
22:             MakeCFEdge(n, n′)
23:         return n
24:     for each s ∈ T do                          ▷ Create successor control-flow nodes
25:         for each P′ ∈ {L ∪ pd, (L \ pd) ∪ {p0}} do      ▷ Compute successor predicates
26:             Let f′, l′ be the first/last instruction of s in
27:                 CFNode n′ = UnFold(f′, l′, ∞, ∅, P′)
28:                 MakeCFEdge(n, n′)
29:     return n
30: procedure UnFoldCFG(G)
31:     Let f, l be the first/last instruction of the entry block of G in UnFold(l, f, ∞, ∅, {p0})
```

4.2 Discussion of the Algorithm

Algorithm 1 consists of two functions: the algorithm's main function UnFoldCFG and the recursive function UnFold, which actually constructs the unfolded CFG. The latter function's parameters f (first) and l (last) represent machine instructions that need to be unfolded next. The integer argument d (delay) is needed to track branch delay slots across control-flow node boundaries. The argument T (targets), likewise, is used to track the set of potential branch targets during the handling of branch delay slots. The function's last argument P (predicates) represents the set of active predicates known to be `true`.

The algorithm starts off by processing the CFG of a function provided by LLVM (l. 31), considering the instructions at the function's entry point, which are not in a branch delay

slot ($d = \infty$) and not executed under any specific predicate condition ($P = \{\texttt{p0}\}$). This triggers the recursive processing of all instructions in the CFG provided by LLVM and the construction of the unfolded CFG. The actual unfolding then proceeds in two steps.

First, all the instructions between the arguments l and f of function UnFold are analyzed (l. 4 – 15) in order to find locations where the control-flow needs to be split. A split may be necessary due to one of the following reasons: (a) the end of the original basic block of LLVM is reached (fall-through), (b) a branch effectively transfers control to another basic block after its branch delay slots, or (c) a predicate definition is encountered.

Each instruction is analyzed in turn. Instructions that are nullified under the current predicate set P (IsNop, l. 7) are ignored. Two instruction classes need special attention, since they may cause a control-flow split: instructions that define a predicate (IsPredDef) and branches (IsBranch). Predicate definitions are handled similar to branches in traditional CFGs and immediately lead to a control-flow split (**break**), remembering the current end location (e) and the newly defined predicate (pd). Branches, on the other hand, may cause a delayed control-flow split (BranchDelay). The variable d tracks the number of remaining branch delay slots (the variable is initialized to ∞ if the analyzed code is not in a branch delays slot). Once this counter reaches 0 the actual control-flow split occurs (l .14). Note that predicate definitions may also appear in branch delay slots. In this case the variable d is passed as an argument to subsequent recursive calls to the function UnFold. Since a branch was encountered, the branch targets also need to be remembered and potentially passed on the recursive calls using variable T. If no control-flow split is encountered by the analysis, i.e., no instruction defines a predicate or branches, the **for** loop terminates normally. This only happens for basic blocks with a fall-through.

The set of live predicates (L) is tracked additionally, while instructions are processed. This set is initialized with the incoming argument P (l. 3) and updated whenever the live range of a predicate ends (l. 5). Note that a location, where the live range of a predicate ends, could be exploited to rejoin the control flow. The algorithm does not take advantage of this and essentially *extends* the live ranges of predicates up to a control-flow split.

The second step of the algorithm (l. 17 – 29) is concerned with the actual construction of the unfolded CFG. After leaving the **for** loop, a new control-flow node is created representing the instructions between f and e that are executed under the predicate set P (l. 16). It remains then to discover and unfold the successor control-flow nodes, depending on the nature of the control-flow split determined during the first step.

Fall-throughs (case a from above) and completed branches (case b) always transfer control to another basic block in LLVM's CFG. The main difference is that no branch target is known for fall-throughs (case a), which can simply be obtained using the function FallThroughTarget (l. 18). The remainder of the processing is identical (l. 24 – 29). Each branch target is analyzed through a recursive call to UnFold, considering the new set of active predicates P' as well as the branch target's first/last instruction. Note that a predicate definition (case c) may coincide with cases (a) and (b). The active predicates are thus computed from the live predicates L and the newly defined predicate pd (l. 25). The targets are then visited by the algorithm with the predicate `true` ($L \cup pd$) and `false` ($L \setminus pd$). Note that `p0` always remains `true` and consequently needs to be readded.

The remaining control-flow splits, that are not covered by the previous paragraph, are due to predicate definitions (case c), which may either occur in the middle of basic blocks or in a branch delay slot. Both situations require a slightly different handling (l. 19 – 23), since control is *not* (yet) transferred to another basic block of LLVM's CFG. The set of active predicates P' is computed as in the regular case. However, the remaining instructions

(NEXT, l. 21) of LLVM's current basic block need to be analyzed after the control-flow split, while remembering potentially ongoing branches. This is accomplished by passing the values of d and T to the recursive invocation of UNFOLD. This allows the recursive invocation to correctly track the branch targets and the number of branch delay slots, i.e., if the split actually occurred in a branch delay slot.

Extensions

The presented algorithm is a somewhat simplified version of the actual implementation. Most notably, the Patmos processor can fetch and issue multiple operations in parallel using instruction bundles. This means that corner cases may arise that need to be considered. For instance, predicate definitions and branches can be combined into the same bundle. The implemented algorithm considers function calls and returns, whose branch delay slots also need to be accounted for. The handling of function calls was omitted for brevity. These extensions slightly complicate the algorithm, but do not impact its overall structure.

Another, more involved extension, is the handling of branches nested within other branch delay slots. The presented algorithm is only correct as long as the predicates of the nested branches are disjoint. This can be handled by replacing the arguments d and T of the function UNFOLD by a stack data structure. This allows to track all executing branches at the same time, detect the completion of a branch, and split the control flow accordingly.

Complexity

The UNFOLD function essentially performs a depth first search on the CFG provided by LLVM. Each instruction is processed once for every set of potentially active predicates, whose number can be bounded by 128 (2^7). Note that `p0` is always `true` and thus cannot impact this bound. The algorithm thus is linear in the number of instructions and control-flow edges.

5 Preliminary Experiments

The following section presents the results from preliminary experiments measuring the overhead induced by unfolding. The implementation is part of the analysis framework of the Patmos compiler, which is based on LLVM 3.5. The unfolded CFG is merely used for analysis purposes and essentially represents an additional annotation layer on top of the data structures of LLVM. The binary code of the analyzed programs is thus not modified. The extended version of the previously described algorithm was applied to a subset of the TACLe benchmarks [6], i.e., those adopted from the MiBench suite. The programs were compiled with optimizations enabled (`-O2`), while varying the issue-width (single-issue vs. VLIW) and the compiler's handling of branch delay slots (non-delayed only, delayed only, mixed). This results in 6 configurations overall. The size of the unfolded CFGs for each of these configurations is compared against the original instruction count in LLVM's CFG.

Figure 2 shows the normalized increase in the number of instructions for the three configurations with VLIW instruction bundles. As can be seen, the overhead induced by unfolding is usually low, ranging between 10% and 20%. The `susan` benchmark is the only exception, showing an increase between 43% and 45%. The if-conversion optimization is particularly effective for this benchmark, covering larger regions and producing more complex predicates. Note that the observed overhead does not come as a surprise. It is well known that the share of conditional branches in typical programs roughly falls into a similar range as the observed overhead. This indicates that, overall, only a few instructions are duplicated

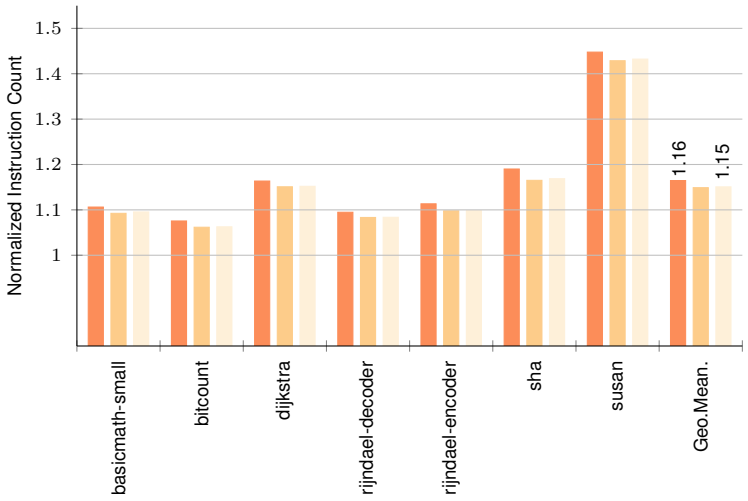


Figure 2 Increase in the number of instructions due to unfolding for the delayed (■), mixed (■), and non-delayed (■) configurations with VLIW instruction bundles, normalized to the size of LLVM's original CFG (lower is better).

by the unfolding algorithm for each computed *condition* – despite the fact that the algorithm artificially *extends* the live ranges of predicates. The configurations for the single-issue processor follow a similar trend (not shown for brevity).

The runtime overhead of the proposed algorithm is negligible and amounts to 0.1*s* on average, which represents 0.9% of the code generation time (excluding other WCET analysis steps). Also note that the unfolded CFG allowed us to improve other analyses. The value range analysis, for instance, is able to take advantage of control-flow splits at predicate definitions as explained in Section 3.

6 Related Work

Predicated computer architectures received considerable attention in the 1990s with the development of VLIW and EPIC architectures that tried to exploit instruction-level parallelism through static compilation techniques rather than hardware [7].

Various compiler optimizations have been developed targeting the transformation of regular code into predicated code [14, 25] and the optimization of predicated code [5, 22]. A common problem for these optimizations is the need to understand the relations between predicates [11, 21], i.e., which predicates can be live at the same time. The underlying machine code may evolve through optimizations in the compiler, which might require these analyses to be performed multiple times. The analyses thus need to be fast and only reason about predicate relations that can be deduced from the *structural* relations between predicates. Information on the actual conditions, e.g., the tested values, are not captured. The work is somewhat orthogonal to our approach and might help to reduce some of the overhead induced by useless code duplications. The techniques are, in addition, concerned with the analysis of the predicates themselves and do not allow to obtain other analysis results.

Hu [9] addressed this issue by refining the semantics of predicated code and redefining several typical concepts used in compilers/static analyzers (e.g., dominance and data dependencies). She also showed how predicate-aware data-flow analysis can be realized using the example of reaching definitions. Similar techniques could be applied to many other analysis

techniques, including those used in typical WCET analyzers. However, this would require a considerable engineering effort in order to adapt all existing analyes accordingly.

The single-path programming paradigm, which can often be found in the context of real-time systems, takes predication to the extreme: (almost) all control-flow is eliminated from the program and replaced by predicated code [15]. The idea is to avoid WCET analysis altogether and instead generate code that exhibits the same execution time under *all* execution conditions. Geyer et al. [8] propose a code generator that is able to produce single-path code from a given input program. The proposed approach heavily relies on predication, which generally leads to constant timing. However, some timing variations can still be encountered due to variations of the program inputs and due to memory accesses to the data cache.

Starke et al. [24] propose a lightweight approach for predicated execution for their time-predictable VLIW. The proposed approach consist of only a single predicate register, which limits the compiler's ability to handle complex predicate expressions and may also impact the attainable instruction-level parallelism. The results indicate that predicated execution is mostly beneficial with regard to the WCET. However, the authors do not describe how the WCET analysis handles predicated operations. The technique proposed in the previous section might allow to further improve the estimation of the WCET for their processor.

7 Conclusion

In this work a lightweight approach to the handling of predicated code in WCET analyzers was presented. Predicate definitions are treated similar to conventional branches and immediately lead to a control-flow split. Subsequent instructions are then analyzed twice, once assuming that the predicate evaluates to `true` and once assuming it evaluates to `false`. The hidden control flow in predicated code is recovered and explicitly represented in an unfolded CFG. The presented algorithm is able to perform the desired control-flow unfolding and keep track of branch delay slots for a simplified single-issue architecture. The actual implementation is able to handle parallel instruction bundles, function calls, and nested delayed branches. Our preliminary evaluation shows that the unfolding does not result in excessive code duplication and yields a moderate code size increase of about 16% on average.

References

1 F. Brandner, S. Hepp, and D. Prokesch. D5.2 – Initial compiler version, 2012. Report of T-CREST Deliverable D5.2, `http://www.t-crest.org/page/results`.

2 P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proc. of the Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, 2014.

3 M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 169–176. IEEE, 2003. `doi:10.1109/EMRTS.2003.1212740`.

4 D. L. Dvorak. NASA study on flight software complexity, 2009. NASA Office of Chief Engineer, Technical Excellence Initiative.

5 A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 180–191. IEEE, 1995.

6 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proc. of the Int'l Workshop on Worst-Case Execution Time Analysis*, volume 55 of *OASIcs*, pages 1–10. Schloss Dagstuhl, 2016. `doi:10.4230/OASIcs.WCET.2016.2`.

7 J. A. Fisher, P. Faraboschi, and Y. Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

8 C. B. Geyer, B. Huber, D. Prokesch, and P. Puschner. Time-predictable code execution – instruction-set support for the single-path approach. In *Proc. of the Int'l Symposium on Object/component/service-oriented Real-time distributed Computing*, pages 1–8, 2013. doi:10.1109/ISORC.2013.6913195.

9 P. Hu. Static analysis for guarded code. In *Proc. of the Int'l Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 44–56. Springer, 2000.

10 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proc. of the Conference on Languages, Compilers and Tools for Embedded Systems*, pages 163–172. ACM, 2013. doi:10.1145/2465554.2465567.

11 R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 100–113. IEEE, 1996.

12 E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. doi:10.1109/TVLSI.2015.2405614.

13 S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proc. of the Architecture of Computing Systems Conference*, pages 122–134. Springer, 2011. doi:10.1007/978-3-642-19137-4_11.

14 J. C. H. Park and M. Schlansker. On predicated execution. Technical report HPL-91-58, HP Laboratories, 1991.

15 P. Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proc. of the IFIP World Computer Congress*, pages 163–172. Kluwer, 2002.

16 J. Reineke. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proc. of the Int'l Conference on Computer Design*, pages 87–93. IEEE, 2012. doi:10.1109/ICCD.2012.6378622.

17 J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.

18 C. Rochange, S. Uhrig, and P. Sainrat. *Time-Predictable Architectures*. ISTE Wiley, 2014. doi:10.1002/9781118790229.

19 S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASIcs*, pages 83–92. Schloss Dagstuhl, 2014. doi:10.4230/OASIcs.WCET.2014.83.

20 M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OASIcs*, pages 11–21. Schloss Dagstuhl, 2011. doi:10.4230/OASIcs.PPES.2011.11.

21 J. W. Sias, W.-M. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 112–123. ACM, 2000. doi:10.1145/360128.360141.

22 M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H.-H. S. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proc. of the Int'l Symposium on Code Generation and Optimization*, pages 169–178. IEEE, 2003.

23 R. A. Starke, A. Carminati, and R. S. De Oliveira. Evaluating the design of a VLIW processor for real-time systems. *ACM Trans. Embed. Comput. Syst.*, 15(3):46:1–46:26, 2016. doi:10.1145/2889490.

24 R. A. Starke, A. Carminati, and R. S. de Oliveira. Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications. *Microprocessors and Microsystems*, 49:1–8, 2017. doi:http://doi.org/10.1016/j.micpro.2016.11.017.

25 A. Stoutchinin and G. Gao. If-conversion in SSA form. In *Proc. of the Int'l Euro-Par Conference*, pages 336–345. Springer, 2004. doi:10.1007/978-3-540-27866-5_43.

Towards Multicore WCET Analysis*

Simon Wegener

AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany
wegener@absint.com

Abstract

AbsInt is the leading provider of commercial tools for static code-level timing analysis. Its `aiT Worst-Case Execution Time Analyzer` computes tight bounds for the WCET of tasks in embedded real-time systems. However, the results only incorporate the core-local latencies, i.e. interference delays due to other cores in a multicore system are ignored. This paper presents some of the work we have done towards multicore WCET analysis. We look into both static and measurement-based timing analysis for COTS multicore systems.

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

Keywords and phrases Worst-Case Execution Time (WCET) Analysis for Multicore Processors, Real-time Systems

Digital Object Identifier 10.4230/OASIcs.WCET.2017.7

1 Introduction

"The problem of determining upper bounds on execution times for single tasks and for quite complex processor architectures has been solved" [23]. While this statement was true a decade ago, when safety-critical embedded systems only used singlecore processors, it no longer holds since the emergence of the multicore processor in the hard real-time context.

The problem for the timing analysis of multicore systems are the interference delays due to conflicting, simultaneous accesses to shared resources, like for example the main memory (see Figure 1). On a singlecore system, the latency of a memory access mostly depends on the accessed memory region (e.g. slow flash memory vs. fast static RAM) and whether the accessed memory cell has been cached or not. On a multicore system, the latency also depends on the memory accesses of the other cores, because multiple simultaneous accesses might lead to a resource conflict, where only one of the accesses can be served directly, and the other accesses have to wait. These interference delays need to be included in a worst-case assessment.

As part of our ongoing work towards multicore timing analysis, we looked in the last few years into methods for multicore timing analysis, strategies to reduce resource conflicts, and COTS multicore architectures. The paper at hand contains the findings of this survey. Additionally, we reviewed the AURIX TC27x and its potential for multicore timing analysis. To the best of our knowledge, such a review of the AURIX TC27x has not been published yet.

This paper is organized as follows: In Section 2, we describe approaches for multicore WCET analysis. In Section 3, we present some strategies to reduce the amount of resource conflicts, which helps to improve the results of a multicore WCET analysis. In Section 4, we

* This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research with the funding ID 01IS16025B. The responsibility for the content remains with the authors.

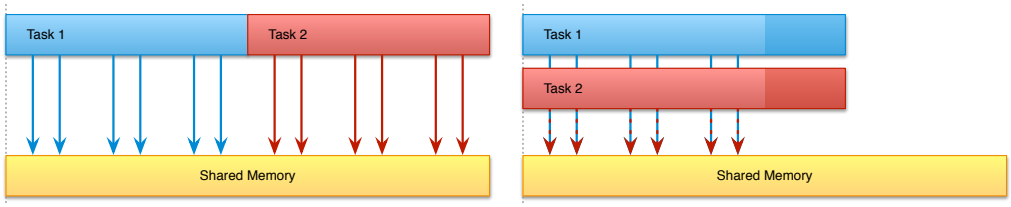


Figure 1 Two tasks which access the shared memory. On a singlecore system (left), each access takes a certain amount of time, but is not delayed due to resource conflicts. On a multicore system (right), resource conflicts happen due to simultaneous accesses to the shared memory. Consequently, the overall execution time of each task is increased.

introduce two multicore architectures and give some hints concerning their suitability for WCET analysis. In Section 5, we list some related work. Finally, in Section 6, we conclude and present some of the future work.

2 WCET Analysis for Multicore Systems

Any sound WCET analysis targeting multicore systems must take the interference delays into account that are caused by resource conflicts. Ignoring these delays might result in underestimation of the real WCET. Assuming full interferences at all times, however, is also not a solution, but might result in huge overestimation. Therefore, the interferences, and consequently, the resource conflicts, have to be analysed in order to get precise results. There a basically two possibilities for such an analysis:

- One can perform a joint analysis of all tasks and cores of the system. This way, the scheduling of the tasks and their allocation to the cores is known to the microarchitectural analysis. While this type of analysis may produce the most precise results, it is often disregarded due to the high computational complexity, rendering this approach infeasible.
- One can first perform separate WCET analyses for each task on each core, ignoring all interferences from the outside. Later, in a second step, the costs due to the interferences are analysed and incorporated into the results from the former analyses. This is the same scheme that is already applied on singlecore systems to derive the worst-case response time which incorporates communication delays and task switch/preemption costs into the WCET bound. Albeit being computationally easier, this approach needs to take extra care for the many non-timing-compositional features of modern processor architectures.

2.1 Static WCET Analysis

For static WCET analysis, we propose to use the second approach to tackle multicore systems, i.e. separate singlecore WCET analysis on the code level and an interference analysis on the system level. For singlecore WCET analysis, AbsInt provides `aiT` [1]. Supported multicore architectures are, among others, the Infineon AURIX TC275 and TC277.

To support interference analysis, we currently implement Worst-Case Resource Access (WCRA) analyses in the `aiT` framework. Here, the microarchitectural analysis is used to determine the maximal number of accesses to a shared resource that can occur during a task's execution. One example for such a resource is the shared memory. This number can be used to estimate the influence of a task on the timing of other tasks.

As always, the more predictable a system is, the easier it is to analyse it statically, and the more precise are the results [6, 8, 24]. This is in particular true for the interference delays.

However, since memory accesses are orders of magnitude slower than normal instructions, one can argue that the pipeline will drain during the processing of memory accesses. Thus, the interference delays imposed by resource conflicts do not cause timing anomalies and can be added later to the singlecore WCET bound [19].

Under this assumption (timing compositionality), singlecore WCET bounds and WCRA bounds can be used to estimate the multicore WCET in a system level analysis. The precise design of such an analysis depends on the type of delay caused by resource contention. An overview on the different kinds of approaches concerning contention on accesses to shared resources is given in [10].

Clearly, the assumption that interference delays do not cause timing anomalies is a rather strong assumption. According to recent research [13], there are two possible ways to gain timing compositionality: sound penalties and compositional base bounds. A sound penalty comprises all direct and indirect effects on the execution time in case of a resource conflict. Unfortunately, there is no known method to compute such sound penalties that generally hold. However, the authors conducted some experiments to find sound penalties for a set of benchmark programs. They observed that the impact of indirect effects increases with shorter memory latencies, because the pipeline is less likely to hide these effects behind long running memory accesses. This observation gives some justification for the above assumption, although no thorough research has been conducted yet to validate it.

The second approach are compositional base bounds. Here, the singlecore WCET bound is augmented with a safe approximation of the possible indirect effects. Then, in the system-level analysis, the direct costs of resource conflicts (the interference delays) are added to the singlecore WCET bound. Since the indirect effects are already incorporated, this gives sound results – even for non-compositional architectures. However, computing the safe approximation of the indirect effects might be computationally expensive. We refer to [13] for the details of this approach.

2.2 Hybrid (Measurement-based) WCET Analysis

In the last years, we also investigated hybrid measurement-based approaches to WCET analysis of multicore systems. For example, we developed together with Accemic and TU Darmstadt a system for the non-intrusive continuous analysis of a dualcore system based on the ARM Cortex-A9 [9].

By its nature, an analysis using measurements to derive timing information is always a joint analysis, because the effects of other running cores are directly visible in the measurements. The quality of the measurements depends on the source of measured events. To avoid the probe effect, one needs to forego software instrumentation. Embedded trace units of modern processors, like Nexus 5001™ [14] or ARM CoreSight™ [5] allow the fine-grained observation of a core's program flow.

When using measurement-based methods, one needs to take care that all possible execution scenarios are observed during the measurements. With end-to-end measurements, this is normally not possible. Instead, one applies hybrid approaches where short snippets are measured and later combined in a static path analysis. The assumption is that it is easier to measure the worst case for each of the snippets the more fine-grained they are. Moreover, when the snippets match the basic block model often used in static WCET analysis, it is easier to perform the path analysis.

For modern Nexus-based ETUs like the one found in the NXP QorIQ P- and T-Series, this is unfortunately not the case. Here, Branch History Messages (BHM) are emitted for indirect branches. The program flow of the taken/not-taken direct branches since the last

BHM is encoded in the message as a sequence of bits. Thus, we do not get timing information for every branch. While the information in the BHM trace is enough to fully reconstruct the control flow, it does not directly map to the basic block model we normally use for path analysis. This is in particular true for small loops consisting of only one or two basic blocks which might be covered by only one BHM although multiple iterations have been executed.

Besides the obvious problem that one needs to generate enough input data to observe all possible (or all important) scenarios for measurement-based approaches, there are (at least) two more multicore-specific problems to consider:

- When multiple cores are running, they also generate a multiple of the trace messages a singlecore system would emit. Thus, the limited bandwidth of the system's trace port might not suffice to transport all trace messages to the debugger. Consequently, some messages are lost (the trace contains "holes") and important timing information cannot be obtained. This leads to lower confidence in the resulting WCET estimate.
- Measurements also observe the timing effects of events like DRAM refreshes and resource conflicts. While this is exactly the reason why we consider measurement-based approaches for multicore timing analysis, this might also lead to huge overestimation. Consider the interference delays induced by the use of a shared interconnect. Assume now that one in ten accesses will suffer a severe interference delay. These delays will possibly occur for any access in the program. During the observation of the program's execution we measure for most accesses both the case where no interference happens as well as the case where the delay occurs. Due to the worst-case assessment, we will incorporate the delay for all these accesses. Thus, we overestimate the real WCET because many more accesses on the critical path will incorporate the delay than the "one in ten" ratio suggests.

Besides their use in WCET analysis, measurements can be used to construct execution time profiles [7]. These are particularly useful for performance analysis, because they also cover the average case.

3 Strategies to Reduce Resource Conflicts

There are several strategies to reduce the amount of resource conflicts, either by controlling when accesses to shared resources happen or by limiting the amount of accesses to shared resources. In the following, we want to present some of them.

3.1 Privatisation of Shared Resources

One idea to avoid resource conflicts is the privatisation of shared resources. This can be done by assigning the resource to a specific core for a limited amount of time. In this timespan, the core has the sole ownership over the resource and no other core is allowed to access the resource.

Schranzhofer et al. presented in [22] a TDMA-based resource scheduling, where each task is split into an input/output phase at the beginning, a computation phase in the middle, and an input/output phase again at the end of its runtime. The tasks on the different cores are then started with an offset in such a way that the input/output phases of the different tasks do not overlap (see Figure 2). This scheduling avoids the resource conflicts altogether, but needs changes to existing code (and maybe operating system (OS) support) in order to be implemented.

Another scheme for resource privatisation was presented in [8]. Here, warm-up and cool-down phases are added to each task (see Figure 3). In the warm-up phase, data is copied

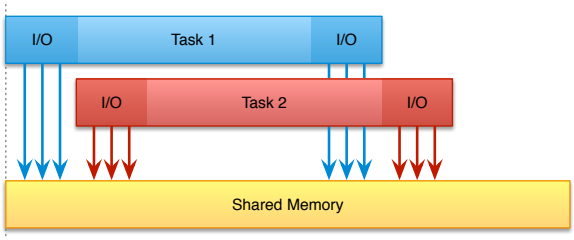


Figure 2 Privatisation of shared resources by applying a TDMA-based scheduling of accesses to shared resources. Resource conflicts are avoided because the input/output phases of different tasks do not overlap.

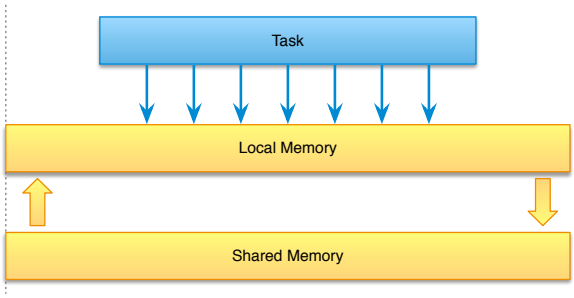


Figure 3 Privatisation of shared resources by copying data from shared to local memory and back again in warm-up and cool-down phases. Resource conflicts are avoided because no accesses happen to shared memory during the task's execution.

from the shared memory to local memory. In the cool-down phase, data is copied from the local memory to shared memory. This approach gives some control when resource conflicts occur, as they can only happen during warm-up and cool-down phases. To completely avoid them, use a TDMA-based scheduling for the warm-up and cool-down phases, similar as above. Naturally, this scheme only works if the target architecture has some local memory that can be used for privatisation. Moreover, tool or OS support is needed to implement the warm-up and cool-down phases. On the other hand, existing code does not need to be changed. However, the scheme may have severe performance overhead.

3.2 Runtime Resource Capacity Enforcement

Instead of avoiding resource conflicts, one can also limit them to an amount such that all tasks still do not miss their deadline, even when we assume full interference for the amount of accesses which equals the aforementioned limit. This concept is called runtime resource capacity enforcement and was presented in [19].

Here, one analyses each task regarding WCET and WCRA (see Section 2.1). Then, one determines how many accesses assuming full interference are allowed for a task t_0 in order to not miss its deadline. This is the amount of resource accesses that all other tasks are allowed to perform during the execution of t_0. Each other task t_x is then assigned its capacity, i.e. its fair share of this amount. A runtime monitoring system observes the number of actual resource accesses of each task. The OS checks that the capacity of a task is not exceeded (and suspends the task otherwise). This scheme is in particular helpful for mixed-criticality scenarios.

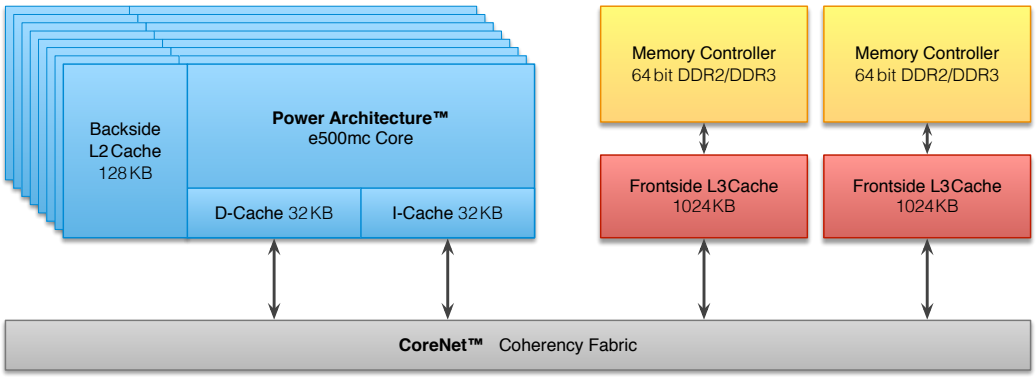


Figure 4 Block diagram of the Freescale P4080.

4 Multicore Platforms

In the following, we assess two different multicore platforms and their suitability for WCET analysis. The Freescale QorIQ P4080 has been used as one of the evaluation platforms in the German research project ARAMiS [3]. It was also the platform of choice for several research papers (e.g. [18, 19]). The Infineon AURIX TC27x is considered as an evaluation platform in the German research project ARAMiS II [4].

4.1 Freescale QorIQ P4080

The Freescale P4080 [11] is a prominent example of a multicore platform where the interference delays have a huge impact on the memory access latencies. Nowotsch et al. [19] measured maximal write latencies of 39 cycles when only one core was active, and maximal write latencies of 1007 cycles when all eight cores were running.

The P4080 consists of eight PowerPC e500mc cores running at 1.5 GHz. The memory hierarchy of the P4080 looks as follows (see also Figure 4): Each core has separate L1 instruction/data caches and a unified L2 cache. The L1 caches have a capacity of 32 KB[1] each, whereas the L2 cache has a capacity of 128 KB. The cores communicate with each other and the main memory over a shared interconnect, the CoreNet Coherency Fabric. Via this interconnect, two memory controllers attach DDR2/DDR3 RAM to the cores. Each of them sits behind 1 MB of unified L3 cache.

The P4080 is a non-compositional architecture according to the classifications in [24] due to its use of PLRU and FIFO replacement policies in caches, translation lookaside buffers, branch target buffers and branch history tables. However, it can be configured in a more predictable way avoiding domino effects:

- Dynamic branch prediction can be switched off.
- TLBs can be preloaded to avoid misses.
- The L1 data cache can be used in write-through mode.
- The L2 cache can be used as scratchpad memory.
- Partial cache locking can be used to gain LRU replacement policy.

[1] For reasons of compatibility with the processor manuals, we use the abbreviation KB for 2^{10} bytes, and MB for 2^{20} bytes.

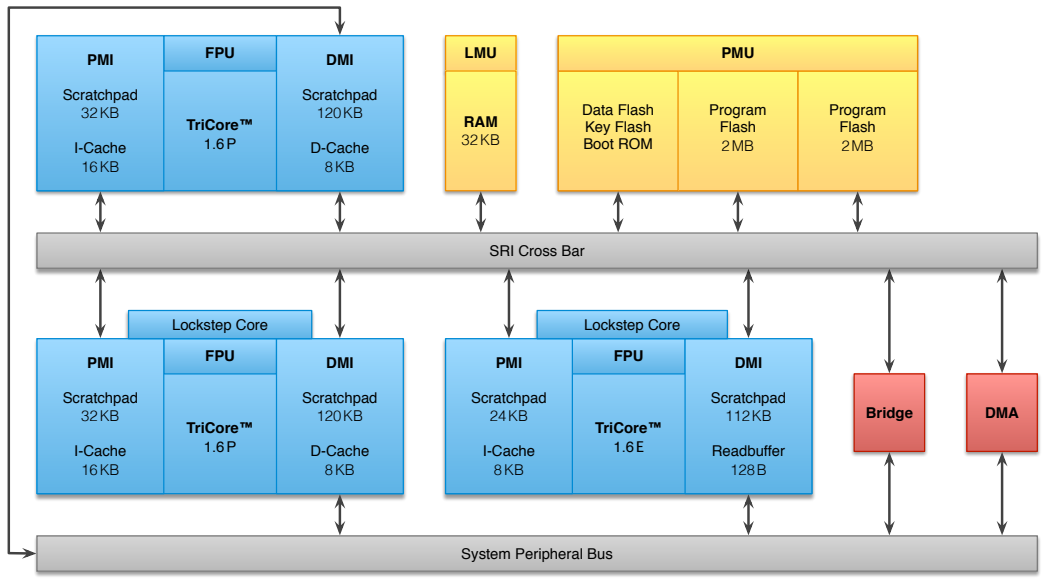


Figure 5 Block diagram of the Infineon AURIX TC27x.

Unfortunately, there is not enough documentation publicly available concerning the CoreNet. Thus, no static analysis predicting its behaviour can be developed. Latencies of memory accesses crossing the interconnect must therefore be obtained by other means, e.g. measurements and incorporated in a response time analysis.

The e500mc core is a complex processor architecture. Constructing and validating a mathematical model suitable for static singlecore timing analysis might take several man-years. `aiT` currently does not support the P4080. However, `aiT` supports the Freescale MPC7448 which has a similarly complex core but is a singlecore architecture.

4.2 Infineon AURIX TC27x

The Infineon AURIX TC27x [15] (Figure 5) is a multicore processor widely used in the automotive domain. It consists of two TC1.6P (performance) cores and one TC1.6E (efficiency) core. One of the performance cores and the efficiency core have attached a checker core such that they can run in lockstep mode. This lockstep mode does not affect the timing behaviour of the system (except in case of disagreement, then a failure is reported to the Safety Management Unit, which may trigger a handler).

Memory and Caches. Each of the performance cores has 120 KB of data scratchpad RAM and 32 KB of program scratchpad RAM. Additionally, each of them has 16 KB of instruction cache and 8 KB of data cache. The efficiency core has 112 KB of data scratchpad RAM and 24 KB of program scratchpad RAM. Additionally, it has 8 KB of instruction cache and a 128 bytes wide read buffer. The caches are organized as two-way set associative caches with LRU replacement strategy. The data caches are write-back caches, but they can be bypassed.

The three cores of the AURIX TC27x are attached via the Program Memory Interface (PMI) to the SRI Cross Bar (SRI), and via the Data Memory Interface (DMI) to the SRI and the System Peripheral Bus (SPB). All peripherals are attached to the SPB except the On-Chip Debug Support (OCDS), which is connected to the SRI via the DMA interface.

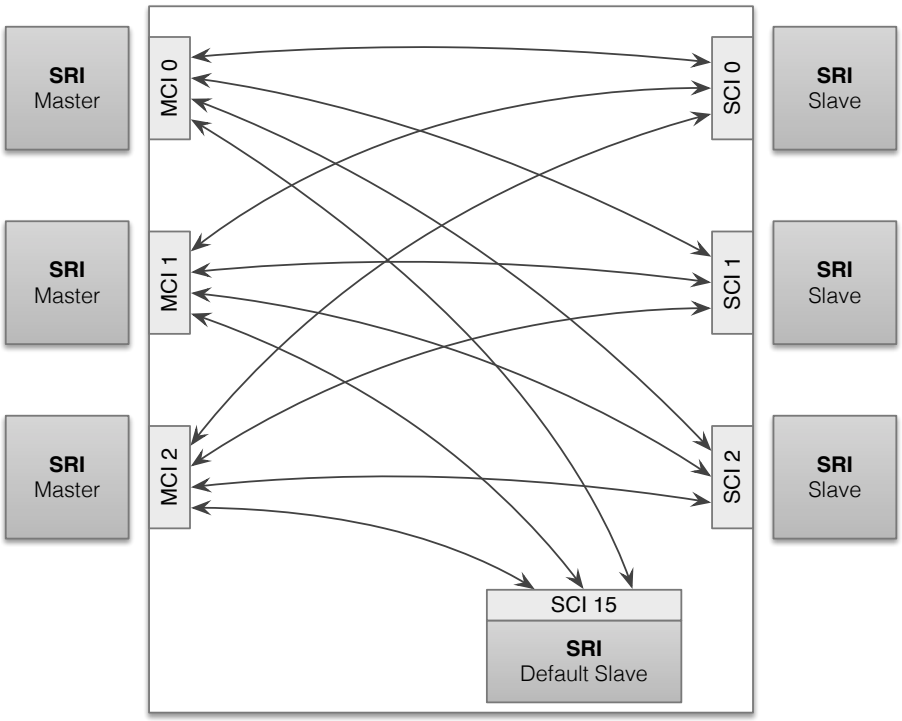


Figure 6 Overview of the structure of the SRI Cross Bar.

Attached to the SRI are also the Local Memory Unit (LMU) and the Program Memory Unit (PMU). The LMU provides 32 KB of shared SRAM to the system, whereas the PMU provides two independent program flash memories (2 MB each) and a data flash memory (464 KB). Each of the flash memories has its own port on the SRI.

SRI Cross Bar

The SRI (Shared Resource Interconnect) Cross Bar connects up to 16 bus masters with up to 15 slaves (and one additional default slave) via point-to-point connections (see Figure 6). The default slave has two purposes. First, it handles the accesses to the SRI configuration registers. Second, the error handling concerning the SRI is done here. The three DMIs and three PMIs are master devices for the SRI, as is the DMA interface. A resource conflict happens when two or more master devices try to access the same slave device. Each slave has its own arbiter to handle these resource conflicts (see Figure 7).

The arbitration rules are as follows: On the top level, the priorities of the master devices decide which request is handled first. The highest priority is 0, the lowest priority is 7. Only one master is allowed to have the same priority, except for the priorities 2 and 5. Here, an additional level of arbitration is performed. All masters with priority 2 form a round robin group, the masters with priority 5 form another one. Within these groups, round robin scheduling is done for arbitration.

Moreover, the arbiters contain a mechanism for starvation prevention. Starvation can happen if some high priority master continuously accesses the same slave such that a master with lower priority never gets its access granted. To prevent this, some kind of priority ceiling is performed when an access is not granted for a configurable timespan.

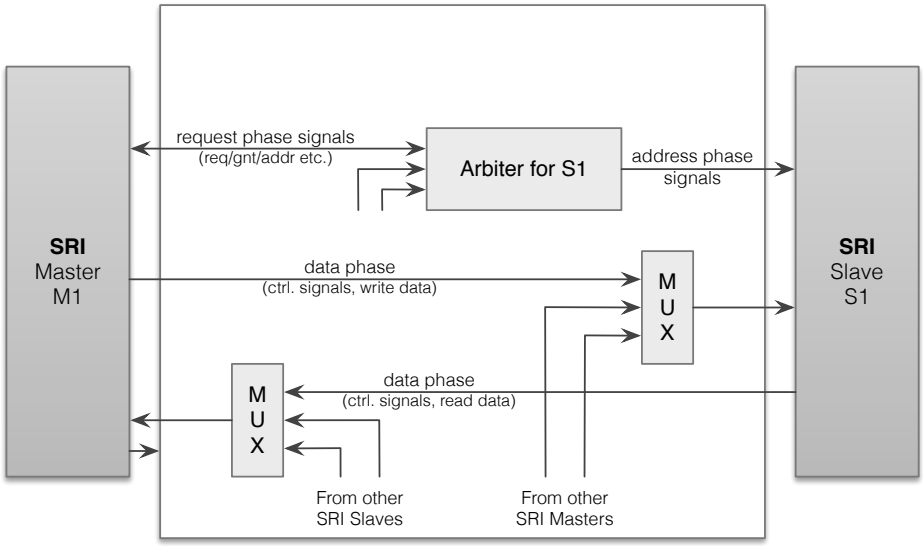


Figure 7 Detailed view on one point-to-point connection in the SRI Cross Bar. Each slave has its own arbiter.

The SRI Cross Bar is well documented (about 70 pages) in the AURIX TC27x user manual [15]. It should be possible to derive the necessary formulas to predict the number of wait cycles depending on the number of conflicting accesses. However, the concrete derivation of these formulas remains future work.

Branch Prediction. Both the TC1.6P and the TC1.6E cores use branch prediction mechanisms to improve the performance. The TC1.6E core uses a static branch prediction scheme with the following rules: Branches in the 16-bit instruction format are predicted taken. Branches in the 32-bit instruction format which point backwards are predicted taken. Branches in the 32-bit instruction format which point forwards are predicted not taken. The TC1.6P core uses a dynamic branch prediction scheme that is not further explained in the manual. A sound static analysis thus has to take both cases – correct prediction and misprediction – into account.

Predictability and Proposed Configuration. Given the fact that the TC1.6P pipeline can execute up to three instructions in one clock cycle, and its use of dynamic branch prediction, the occurrence of timing anomalies is likely. However, we assume that the AURIX is a compositional architecture with constant-bounded effects [24].

`aiT` supports the Infineon AURIX TC27x and can be used to compute singlecore WCET bounds for each of the three cores. The only thing that is not supported by `aiT` are the write-back caches. Hence, the data caches need to be bypassed. Moreover, the analysis model of aiT assumes that no other SRI master devices besides the analysed core access the same slaves, i.e. that no resource conflicts happen in the SRI Cross Bar.

In the following, we propose a configuration to minimise the amount of resource conflicts when used in a multicore scenario:

- Use one dedicated program flash memory for each of the performance cores to avoid conflicting accesses. Use the data flash for the efficiency core, if needed.

- Use the core-local data scratchpad whenever possible instead of the shared RAM to reduce conflicting accesses. If possible, preload data from the shared RAM and data flash to the local scratchpad memories to control when accesses to the shared memory happens.
- Place the stack in the core-local data scratchpad.
- Do not access the core-local scratchpad memories from other cores.
- I/O channels (CAN, FlexRay, ...) should not be accessed by multiple cores. Assign each I/O channel in use to a specific core.

5 Related Work

WCET Analysis for multicore systems is an active area of research. At least 17 publications concerning multicore timing analysis have been presented at the past five instances of the International Workshop on Worst-Case Execution Time Analysis. These cover all areas of timing analysis, e.g. static approaches [12], measurement-based approaches [17], resource arbitration [16], hardware [21], mixed-criticality [2] and response time analysis [20].

6 Conclusion and Future Work

In this paper, we presented some of the work that has been carried out at AbsInt to provide tool support for multicore WCET analysis. We currently focus on WCRA analysis to estimate the influence of resource conflicts on the timing of a task, and on BHM traces for hybrid measurement-based timing analysis. Moreover, we gave hints how the Freescale QorIQ P4080 and the Infineon AURIX TC27x, two popular multicore architectures, can be used for real-time embedded systems. In the course of ARAMiS II, we want to evaluate our newly developed analyses in industrial multicore scenarios.

Acknowledgements. I want to thank Michael Schmidt for providing me with the neatly drawn pictures, and Gernot Gebhard and Markus Pister for their valuable comments regarding the AURIX TC27x and the P4080.

References

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

2 Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, and Christine Rochange. WCET and Mixed-Criticality: What does Confidence in WCET Estimations Depend Upon? In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 65–74, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2015.65`.

3 ARAMiS. `http://www.projekt-aramis.de/`.

4 ARAMiS II. `https://www.aramis2.org/`.

5 ARM Ltd. CoreSight™ Program Flow Trace™ PFTv1.0 and PFTv1.1 Architecture Specification, 2011. ARM IHI 0035B.

6 Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 13(4):82:1–82:37, 2014. `doi:10.1145/2560033`.

7 Thomas Ballenthin, Boris Dreyer, Christian Hochberger, and Simon Wegener. Hardware Support for Histogram-based Performance Analysis of Embedded Systems. In *20th IEEE International Symposium On Real-time Computing (ISORC 2017)*. IEEE, 2017. (accepted).

8 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingenieurs de l'Automobile*, 807:26–42, 2010.

9 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2016.4.

10 Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 31–42, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2014.31.

11 Freescale Semiconductor, Inc. QorIQ™ P4080 Communications Processor Product Brief, Rev. 1, 2008. http://cache.freescale.com/files/32bit/doc/prod_brief/P4080PB.pdf.

12 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward Static Timing Analysis of Parallel Software. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 38–47, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2012.38.

13 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real Time and Networks Systems*, October 2016. URL: http://embedded.cs.uni-saarland.de/publications/EnablingCompositionalityRTNS2016.pdf, doi:10.1145/2997465.2997471.

14 IEEE-ISTO. IEEE-ISTO 5001™-2012, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface, 2012.

15 Infineon Technologies AG. AURIX™ TC27x D-Step 32-Bit Single-Chip Microcontroller User's Manual V2.2 2014-12, 2014.

16 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2013.1.

17 Leonidas Kosmidis, Davide Compagnin, David Morales, Enrico Mezzetti, Eduardo Quinones, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. Measurement-Based Timing Analysis of the AURIX Caches. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2016.9.

18 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In Cristian Constantinescu and Miguel P. Correia, editors, *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 132–143. IEEE

Computer Society, 2012. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6214741, doi:10.1109/EDCC.2012.27.

19 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118. IEEE Computer Society, 2014. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6909098, doi:10.1109/ECRTS.2014.20.

20 Dumitru Potop-Butucaru and Isabelle Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–31, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2013.21.

21 Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A Time-Predictable Memory Network-on-Chip. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2014.53.

22 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing predictability on multi-processor systems with shared resources. In *Workshop on Reconciliating Predictability and Efficiency at EMSOFT 2009*, 2009.

23 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.

24 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

The Heptane Static Worst-Case Execution Time Estimation Tool

Damien Hardy[1], Benjamin Rouxel[2], and Isabelle Puaut[3]

1 **University of Rennes 1, Rennes, France**
`damien.hardy@irisa.fr`
2 **University of Rennes 1, Rennes, France**
`benjamin.rouxel@irisa.fr`
3 **University of Rennes 1, Rennes, France**
`isabelle.puaut@irisa.fr`

Abstract

Estimation of worst-case execution times (WCETs) is required to validate the temporal behavior of hard real time systems. Heptane is an open-source software program that estimates upper bounds of execution times on MIPS and ARM v7 architectures, offered to the WCET estimation community to experiment new WCET estimation techniques. The software architecture of Heptane was designed to be as modular and extensible as possible to facilitate the integration of new approaches. This paper is devoted to a description of Heptane, and includes information on the analyses it implements, how to use it and extend it.

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

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

Digital Object Identifier 10.4230/OASIcs.WCET.2017.8

1 Introduction

Knowing task worst-case execution times (WCET) is of prime importance for the timing analysis of hard real-time systems. Timing analysis of multi-task software is in general made of two levels: *WCET analysis* and *schedulability analysis*. WCET analysis estimates the worst-case timing requirements of an isolated task. At this level, activities other than ones related to the considered task (interrupts, blocking, pre-emptions or any kind of interference from other tasks in the system) are ignored. At the *schedulability analysis* level, the analysis considers multiple tasks executing on the processor and competing for resources, and thus may block while attempting to access the resources.

WCETs may be obtained using *static analysis* techniques, *measurement-based* techniques or hybrid techniques (see [29] for a survey). A static WCET analysis tool provides an upper bound (WCET estimate) on the time required to execute a given code on a given hardware without program execution. A static WCET analysis tool should be able to work at a high level to determine the longest path in a code. It should also work at low-level (hardware-level), to capture the worst impact of the target processor on timing. Static WCET analysis is complicated due to the presence of architectural features that improve the processor performance: instruction and/or data caches, branch prediction and pipelines for example. Precisely modeling these architectural features is the key to have precise WCET estimates.

A number of static WCET analysis tools exist. The objective of this paper is to give a high-level and up-to-date view of the static WCET analysis tool we have designed and maintained over the years, named Heptane[1]. The first version of Heptane was developed in the late nineties during the PhD thesis of Antoine Colin. That first version, described in [6] was a research prototype written in OCaml, and originally implemented *tree-based* WCET calculation. At that time, it included the analyses required to obtain WCETs for a Pentium 1 processor (cache analysis, pipeline analysis, branch prediction analysis). Heptane was re-developed in 2003 to have a cleaner software architecture, use C++ instead of OCaml, and support more target processors. This second version [29] implemented both *tree-based* and IPET calculation. This version was used up to 2010 to implement all research related to WCET estimation in our group (compiler-directed branch prediction, cache locking, scratchpad management, analysis of data caches and shared caches in multi-cores, etc.). We then decided to change our development strategy to ease code readability and development of new analyses. We selected to integrate in the main branch of the tool only a minimum number of robust analyses. At that point in time, the software architecture of Heptane was refined again based on our past experience. This paper describes the last version of Heptane.

The aim of Heptane is to produce upper bounds of the execution times of applications. It targets applications with hard real-time requirements (automotive, railway, aerospace domains). Heptane computes WCETs using static analysis at the binary code level. It includes static analyses of micro-architectural elements such as caches and cache hierarchies.

Heptane is an open source software program available under GNU General Public License v3[2]. Heptane is now a reliable research prototype, developed in C++ (approximately 13,000 lines of code) and supports MIPS and ARM v7 instruction sets. In particular, we are using a continuous integration framework to check that the tool builds correctly and perform non regression testing, for the supported target processors and host operating systems. Heptane was demonstrated during the 1st Tutorial on Tools for Real-Time Systems [28].

As compared to other open-source static WCET analyzer tools, Heptane has a special focus on cache analysis (analysis of cache hierarchies, support for multiple replacement policies), but currently supports only two target processors. OTAWA [20] supports more processor architectures than Heptane but implements less advanced cache analyses. SWEET [26] focuses on flow fact analysis and does not include any hardware-level analysis. Bound-T [4] supports different processor architectures but does not provide a cache analysis. Platin is a static WCET estimation tool dedicated to the analysis of the Patmos architecture [24]. Chronos [5] implements advanced low-level analyses, but is limited to the SimpleScalar architecture.

The commercial tool aiT [2] is the most advanced static WCET analysis tool, but unfortunately is not open-source and as such cannot be extended for research studies that require the tool to be modified.

The remainder of the paper provides more details on Heptane and is organized as follows. An overview of Heptane is presented in Section 2. Section 3 describes the WCET estimation techniques implemented in Heptane. The usage of Heptane is detailed in Section 4. Some number to evaluate the performance of Heptane are given in Section 5. Some hints to extend Heptane are mentioned in Section 6. Finally, conclusions and plans for future extensions of the tool are given in Section 7.

[1] The web site of Heptane is: `https://team.inria.fr/pacap/software/heptane/`.

[2] Heptane is registered with APP (Agence de Protection des Programmes) under number IDDN.FR.001.510039.000.S.P.2003.000.10600.

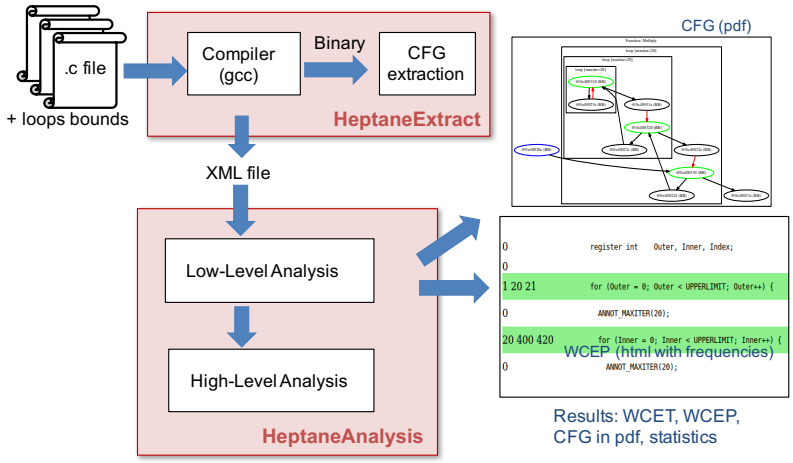


Figure 1 Heptane toolchain: from source code to WCET estimate.

2 Overview of Heptane

2.1 The Heptane toolchain

As illustrated in Figure 1, Heptane is divided in two parts: *HeptaneExtract* for Control Flow Graph (CFG) extraction and *HeptaneAnalysis* for the actual WCET estimation.

HeptaneExtract generates a control flow graph (CFG) from a set of program source-code files written in C or assembly language. It calls the compiler and linker to generate the binary file and then construct the CFG. After the construction of the CFG, *HeptaneExtract* identifies the different loops, attaches the loop bounds information provided in the source file and attaches the instruction addresses based on the binary file. The CFG is stored in XML format to be used by *HeptaneAnalysis* or be manually inspected if needed.

HeptaneAnalysis applies low-level and high-level analyses to produce the WCET estimate. *HeptaneAnalysis* can also be used to generate extra information like a graphical representation of the CFG, a HTML version of the source that highlights the lines on (one of) the longest execution path(s) and some statistics of the application under study.

The CFG produced by *HeptaneExtract* will be gradually enriched by every analysis of *HeptaneAnalysis* with the analysis results. For that purpose, a library named *cfglib* has been developed. This library is designed to manage an extensible program representation that includes objects for programs, functions, loops, basic blocks, edges, instruction, etc. The library also provides so-called *attributes* that can be attached to any of the objects that represent the program. The library provides a small number of built-in attribute types (string, integers, floats, etc.), and new attribute types can be defined for the purpose of an analysis. When attaching an attribute to an object (e.g. a loop), an analysis developer calls the attribute attachment function of *cfglib* for the corresponding attribute type (e.g. integer) and provides as inputs the name of the attribute to be attached (e.g. “maxiter”) and the corresponding value (e.g. 10). The library includes serialization and deserialization facilities for objects and built-in attribute types. As explained later, each analysis may export its results in XML.

2.2 Source code constraints

Since Heptane does not include any analysis of maximum numbers of loop iterations, the source code has to be augmented by the user with annotations to provide loop bounds. At the beginning of each loop body, a dedicated macro named ANNOT_MAXITER has to be inserted (see Listing 1). The macro is defined in the header *"annot.h"* that has to be included. Loop bounds are *local* loop bounds (maximum number of iterations for each entry in the loop) and should be constant values. The macro expands to assembly code, that will create a specific section in the final binary with the maximum number of iterations.

```
#include "annot.h"
int i,j;
for (i = 0; i < 20; i++) {
     ANNOT_MAXITER(20);
     for (j = 0; j < 10; j++) {
          ANNOT_MAXITER(10);
          ...
          }
}
```

Listing 1 ANNOT_MAXITER usage.

Furthermore, some restrictions on the C code are required to be able to generate the CFG statically:

- Pointers to functions are not supported to be able to generate the call graph.
- Indirect jumps and jump tables are not supported. They can be generated for instance by switches constructs.
- Each loop must have a single entry. The identification of loops in Heptane uses DJ Graphs [25]
- Pointer arithmetic is not supported by the data address analysis

Programs not meeting these restrictions are detected and reported as errors. Finally, to be able to perform a correct matching between the loop bounds defined in the source file and the binary file the use of compiler optimization should not affect this matching. It is the user responsibility to select the compiler optimizations that do not change loop bounds.

3 WCET estimation in Heptane

Static WCET estimation methods are generally divided into two steps, commonly named *high-level* analysis and *low-level* analysis. The high-level analysis determines the longest execution path among all possible flows in a program. The low-level analysis is used to account for the processor microarchitecture. In *HeptaneAnalysis*, each analysis is *contextual*, meaning the analysis of a function is performed for every call path of the function (*e.g. main* $\rightarrow g \rightarrow f$ and *main* $\rightarrow f$ for a function f called directly by function *main* and also called by g that is called by *main*).

3.1 High-level analysis

For the *high-level* analysis, *HeptaneAnalysis* implements the most prevalent technique, named IPET for *Implicit Path Enumeration Technique*[17]. IPET is based on an Integer Linear Programming (ILP) formulation of the WCET calculation problem. It reflects the program structure and the possible execution flows using a set of linear constraints. An upper bound of the program's WCET is obtained by maximizing objective function

$\sum_{i \in BasicBlocks} T_i * f_i$. T_i (constant in the ILP problem) is the timing information of basic block i. T_i integrates the effects of micro-architecture, and is determined by the low-level analysis.

The variable f_i in the ILP system, to be instantiated by the ILP solver, corresponds to the number of times basic block i is executed. The values of all variables f_i, once set by the ILP solver to maximize the objective function, identify a set of paths in the program leading to the estimated WCET.

3.2 Low-level analysis

For the *low-level* analysis, *HeptaneAnalysis* implements a *data address analysis*, a *cache analysis* and a *pipeline analysis*.

Data address analysis

The *data address analysis* conservatively determines the addresses of referenced data. In case the exact address of the referenced data cannot be determined, a range of addresses is conservatively provided. The addresses of instructions are determined during the CFG extraction thanks to the addresses present in the binary file.

Heptane includes a *stack analysis*, that calculates the range of addresses for every stack frame, in any call context. The analysis assumes an acyclic call graph, which is common in real-time systems because recursion raises predictability issues. The analysis relies on the knowledge of the base of the stack based address, given as input to *HeptaneAnalysis*, and the size of all functions' stack frames, obtained by scanning the first instruction of every function, responsible for allocating the stack frame. The address of every stack frame is obtained by propagating the address of the stack frame for every callee of every function along the acyclic call graph.

Data address analysis is implemented by an inter-procedural data flow analysis based on abstract interpretation, that evaluates the contents of every register before and after each instruction. Since Heptane does not currently support pointers, it is sufficient to analyze register contents to compute the addresses of load and store instructions and thus no analysis of memory contents is required. The possible abstract values for a register are: $\perp$, $\top$ or an interval of addresses. Value $\perp$ represents an invalid register content; it is used as an initial register value for the first instruction of every basic block. Value $\top$ represents a correct but unknown value (any possible value but $\perp$). We use $\top$ to specify the contents of a register after a load from memory because memory contents are not analyzed. Data flow equations, detailed in [9] for an early version of the data address analysis, are defined for all instructions, to specify the impact of the instruction on the registers abstract values. When the address of a static variable is loaded into a register (e.g. when loading the start address of an array in a register), the symbol table is used to determine the corresponding address interval; for local data the entire stack frame is used as interval.

Cache analysis

The *cache analysis* is the most developed analysis in *HeptaneAnalysis*. It allows to analyze set-associative[3] instruction and data cache hierarchies under different cache replacement policies (LRU, PLRU, FIFO, MRU, Random).

[3] Direct mapped caches and fully-associative caches are specific cases of set-associative caches.

The cache analysis of a cache level in *HeptaneAnalysis* is an implementation of the *Must*, *May* and *Persistence* analyses [27, 13] based on abstract interpretation [7]. The principle of the analyses is to statically associate a Cache Hit/Miss Classification (CHMC) for every memory reference. The CHMC defines for each reference its worst-case behavior with respect to the cache under analysis (e.g. the CHMC is set to *always-hit* only when it is guaranteed the reference will always result in a cache hit, regardless of the execution path followed at run-time).

The *Must*, *May* and *Persistence* analyses calculate for every basic block and every call context a corresponding *Abstract Cache State (ACS)* whose semantics depend on the analysis. For example, the ACS for the *Must* analysis at a given program point contains the addresses of the memory blocks that are guaranteed to be in the cache at that point. The structure of ACS depends on the cache replacement policy. For the most predictable replacement policy LRU [23], the associativity of the ACS is the same as the one of the concrete cache. For LRU replacement, the position of a memory block in a set and a dataflow equations depends on the type of analysis (*Must*, *May* and *Persistence*); for example, for the *Must* analysis, a memory block in the *Must* ACS has an age that is higher than or equal to the age of the block in the LRU stack.

To analyze cache replacement policies other than LRU, we have implemented a method using the metrics proposed in [23] that characterize the life time of references in a cache for different replacement policies. These metrics are used to set the associativity of the ACS to a value lower than the associativity of the concrete cache state for the *Must* and *Persistence* analyses. For example, for a cache with a random replacement policy, any memory block in a set may be replaced upon cache replacement, making such caches equivalent to a direct-mapped cache for the *Must* and *Persistence* analyses regarding CHMC classification.

To analyze cache hierarchies, we have implemented the method proposed in [10, 14] that introduces in the cache analysis a *Cache Access Classification* (CAC) to take into account the filtering effect of the previous cache level in the hierarchy.

For data caches, we assume a *write-through no-write-allocate* policy. A *write-through* policy was analyzed because it is easier to analyze than a *write-back* strategy (in contrast to *write-back* caches, it is easy to know for *write-through* caches when memory accesses will take place). A *no-write-allocate* policy is assumed because *write-through no-write-allocate* is the most common configuration found for *write-through* caches.

Pipeline analysis

The pipeline analysis for all supported architectures, currently considers a simple in-order pipeline free from timing anomalies, [18] with one cycle per stage except for the fetch and memory stages where the results of the cache analysis are taken into account.

Interactions between analyses

The initial CFG is gradually enriched with the results of the analyses, thanks to the attribute attachment facilities provided by the *cfglib* library. For instance, the instruction cache analysis attaches to every instruction and cache level an attribute defining the instruction CHMC and CAC. The attribute will be used by the pipeline analysis to compute the worst-case execution time of every basic block, that will be attached as an attribute to every basic block. The WCET calculation phase will use WCETs of basic blocks to compute the overall WCET. Every analysis has a dedicated method that checks the presence of its inputs (i.e. the corresponding attributes were attached by the analyses executed before). Checking is based

on attribute names. For example, the WCET calculation analysis checks that all loops have an attribute named *maxiter* attached.

4 Using Heptane

Heptane can be installed on Linux and Mac OS X. For its installation, Heptane has some requirements: *libxml2* and an ILP solver (*lpsolve* or IBM CPLEX). To use the optional analyses of Heptane described at the end of this section (graphical output, documentation extraction), *dot*, *epstopdf* and *Doxygen* have to be installed as well. The rest of this section gives an overview of how to use *HeptaneAnalysis*, after CFG extraction (complete documentation can be found in the Heptane web page).

HeptaneAnalysis takes as parameter an XML configuration file describing the architecture under analysis and the analyses to be applied. The analyses operate on a CFG produced by the CFG extractor *HeptaneExtract*. The extracted CFG, whose format is not presented here for space considerations, is a human-readable file with one XML tag per object (program, function, basic block, edge, loop, instruction) with attributes attached when relevant (loop bounds, instruction addresses, etc). A unique identifier is assigned to each object as an XML attribute for deserialization.

4.1 Description of the architecture under analysis

In case the user wishes to use custom parameters describing the architecture or the analyses to perform, a template of configuration file is available in the *config_files* directory. The configuration file starts with an INPUTOUTPUTDIR XML tag to set the working directory:

```
<INPUTOUTPUTDIR name="HEPTANE_ROOT/benchmarks/simple"/>
```

This directory should contain the XML files resulting from CFG extraction and, will contain the result files of the analyses. Then the processor type, the cache hierarchy and the memory have to be described as follows:

```
<ARCHITECTURE>
  <TARGET NAME="MIPS" ENDIANNESS="BIG"/>
  <CACHE nbsets="32" nbways="2" cachelinesize="32"
  replacement_policy="LRU" type="icache" level="1" latency="1"/>
  <CACHE nbsets="64" nbways="8" cachelinesize="64"
  replacement_policy="LRU" type="icache" level="2" latency="10"/>
  <CACHE nbsets="32" nbways="2" cachelinesize="32"
  replacement_policy="LRU" type="dcache" level="1" latency="1"/>
  <CACHE nbsets="64" nbways="8" cachelinesize="64"
  replacement_policy="LRU" type="dcache" level="2" latency="10"/>
  <MEMORY load_latency="100" store_latency="100"/>
</ARCHITECTURE>
```

The user has to provide the target processor (i.e. MIPS or ARM) and the architecture endianness, the memory latency for the loads and stores in processor cycles and for each level of cache:

- the number of sets, the number of ways (associativity), the size of a cache line in Bytes;
- the replacement policy (LRU, PLRU, MRU, FIFO or RANDOM):
- the type of the cache (icache/picache for an instruction cache and dcache/pdcache for a data cache). Letter *p* indicates that the cache is a perfect cache that always hits. In this case, only one cache level can be specified and the other parameters except the hit latency are ignored.

- the level in the hierarchy (1 for L1, 2 for L2 and so on)
- the hit latency in processor cycles

4.2 Using the main analyses

The main part of the configuration file is the description of the analyses the user wants to perform. They have to be defined inside an ANALYSIS XML Tag and they are applied in their order of appearance in the configuration file. In case an analysis A relies on the results of previously applied analyses, analysis A checks the presence of all its mandatory input information. Each analysis has three common parameters:

- *keepresults* set to true to keep CFGs and the results of the analysis in memory and *false* otherwise
- *input_file* set to empty string in case the user wants to reuse the results of a previous analysis, or to a XML file previously exported by an analysis,
- *output_file* set to empty string in case the user does not want to serialize the modifications done by the analysis or to *file.xml* to store it in the mentioned file

The first analysis to apply is to fix the entry point of the program and to compute the calling context of functions accordingly. This is illustrated below:

```
<ENTRYPOINT keepresults="on" input_file="simple.xml" output_file=""
 entrypointname="main"/>
```

In case the user wants to perform a data cache analysis, the data address analysis has to be performed beforehand, to determine the memory addresses of load and store instructions. This analysis takes as parameter the address of the stack pointer before the execution of the entry point, as shown below:

```
<DATAADDRESS keepresults="on" input_file="" output_file="" sp="7FE000"/>
```

For the cache analysis, an analysis has to be defined for each cache[4] of the architecture, and the different levels of caches have to be analyzed in order (i.e. L1 before L2 and so on). The name of the analyses are ICACHE and DCACHE for an instruction cache and a data cache respectively. The analysis has to indicate the cache level and if the *must*, *persistence* and *may* analyses has to be performed, as shown below for a L1 instruction cache:

```
<ICACHE keepresults="true" input_file ="" output_file =""
 level="1" must="on" persistence="on" may="off"/>
```

The pipeline analysis can be called once the caches have been analyzed only. The analysis is defined by a PIPELINE XML tag, with only the common parameters. Finally, WCET estimation is performed by the IPET analysis:

```
<IPET keepresults="on" input_file ="" output_file =""
 solver="lp_solve" attach_WCET_info="true" generate_node_freq="true"/>
```

The declaration should provide the name of the ILP solver (i.e. *lp_solve* or *cplex*). Furthermore, the user should indicate if the estimated WCET of the program as well as the estimated frequency of each basic block have to be kept for the next analyses.

[4] In case of a perfect cache, the analysis has to be performed as well and the corresponding cache level has to be set to 1.

4.3 Other useful analyses

HeptaneAnalysis can provide useful feedback to the user through additional analyses. The first one is the cache statistics analysis, that provides for each cache the number of references, the number of hits and misses along the longest path identified by IPET. The analysis is defined by a CACHESTATISTICS XML tag with only the common parameters.

The user may want to have a graphical representation of the program. *HeptaneAnalysis* can generate a *pdf* file representing the CFG of the benchmark by inserting a DOTPRINT XML tag with only the common parameters.

The user can also print in text format the estimated WCET, the CFG, the call graph, and the loop structure and loop bounds of the benchmark. This is achieved by the SIMPLEPRINT analysis:

```
<SIMPLEPRINT keepresults="on" input_file="" output_file=""
 printWCETinfo="on" printcfg="off"
 printcallgraph="off" printloopnest="off"/>
```

Finally, *HeptaneAnalysis* offers the possibility to display the path identified by IPET as the longest path on the source code, with the estimated execution frequency of each line of code in a HTML file (several times in presence of different calling contexts). To do so, a mapping between the source and the binary[5] is first performed by calling the *addr2line* tool on the binary. In the configuration file, this is performed in two steps: (*i*) the mapping and (*ii*) the production of the HTML file:

```
<CODELINE keepresults="on" input_file="" output_file=""
 binaryfile="simple.exe"
 addr2lineCommand="HEPTANE_ROOT/CROSS_COMPILERS/MIPS/bin/mips-addr2line"
/>
<HTMLPRINT keepresults="on" input_file ="" output_file =""
 colorize="true" html_file="simple.html"/>
```

5 Performance of Heptane

Table 5 gives the analysis time in seconds (total and per analysis) and WCET in cycles for the set of Mälardalen benchmarks[6] that are supported by Heptane. The analysis was performed on MIPS code, with two levels of instruction and data caches with the same structure (2-way set-associative cache with 32 sets and 32-byte blocks for L1, 8-way set-associative cache with 64 sets and 64-byte blocks for L2). The analysis was executed on an Intel Core i7 quad-core . The solver used by the IPET analysis is lp_solve 5.5.2.

The analysis time of all benchmarks except the biggest ones (*nsichneu* and *statemate*) is very low (below one second and up to 3.1 seconds for *fft*). The worst-case analysis time was observed on *nsichneu* and was clearly dominated by the cache analysis time.

6 Extending Heptane

The software architecture of Heptane was designed to be modular and extensible. In addition, it was decided to keep in the main branch of Heptane only the analyses that we think can be useful to a wide audience. Other analyses, developed over the year to

[5] The benchmark has to be compiled with the *-ggdb gcc* option.
[6] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Table 1 Analysis time (seconds) per analysis steps and resulting WCET (cycles).

benchmark	icache (L1)	icache (L2)	dcache (L1)	dcache (L2)	IPET	Total	WCET
bs	0.005	0.009	0.007	0.010	0.005	0.079	4020
bsort100	0.010	0.016	0.009	0.014	0.006	0.150	5812114
crc	0.038	0.049	0.034	0.058	0.013	0.630	1782419
expint	0.012	0.024	0.012	0.020	0.008	0.224	647343
fft	0.261	0.297	0.136	0.175	0.063	3.144	1253683
fibcall	0.002	0.004	0.003	0.005	0.003	0.050	14023
insertsort	0.003	0.006	0.004	0.008	0.004	0.090	52355
jfdctint	0.017	0.019	0.022	0.023	0.011	1.026	100331
lcdnum	0.074	0.116	0.021	0.057	0.009	0.366	9106
ludcmp	0.068	0.085	0.324	0.283	0.016	1.289	381353
matmult	0.014	0.023	0.017	0.027	0.008	0.317	1795585
minver	0.039	0.069	0.044	0.060	0.020	2.784	66835
ns	0.007	0.013	0.011	0.015	0.005	0.153	146208
nsichneu	18.314	21.468	2.815	3.250	0.874	52.775	515015
qurt	0.070	0.089	0.059	0.080	0.027	1.599	111554
select	0.068	0.073	0.023	0.034	0.014	0.547	91798
sqrt	0.012	0.016	0.010	0.018	0.011	0.183	20159
statemate	3.563	5.643	0.322	0.391	0.141	10.730	229575
ud	0.042	0.068	0.096	0.101	0.015	0.674	210770

experiment research techniques are kept in dedicated folders (non exhaustively: analysis of shared cache interference [8, 21], analysis of cache hierarchy management policies [11], cache related preemption delay estimation [19, 22], cache-partitioning [15], traceability of flow information [16], static probabilistic WCET analysis [1, 12] analysis of code caches in just-in-time compilers [3]).

Each analysis is located in a distinct directory and inherits from an *Analysis* base class such that the developer of a new analysis respects good practice (checks the presence of the inputs required by the analyses and cleans up internal attributes). A dummy analysis *DummyAnalysis* is provided as an example. Moreover, the *cfglib* library, central for implementing analyses, is well documented.

7 Conclusion and future work

We have described in this paper Heptane, an open-source software program for WCET estimation, that estimates WCETs from program binaries. Heptane has reached over the years sufficient reliability to be used by researchers external to our group. The most advanced analyses integrated in Heptane are the static cache analyses (support for multiple replacement policies and cache hierarchies). Its weaknesses, that we hope to address in the future, are its lack for automatic extraction of loop bounds, its small number of target processors, and its pessimistic albeit safe data address analysis. Our current work on the tool is to address the latter issue, in particular for stack-allocated data and accesses to arrays. Another direction is to use the XML format used by our open-source CFG management library *cfglib* as an exchange format between tools, to gather the best analysis techniques from different groups into a common WCET estimation infrastructure.

Acknowledgments. Heptane is a collective work to which many people beyond the authors of this paper have participated. The authors would like to thank the following people who actively participated in the development and test of the current version of Heptane: Thomas Piquet, Benjamin Lesage, Erven Rohou, François Joulaud, Nicolas Kiss and Loïc Besnard, as well as developers of older versions, who helped setting up a satisfactory software architecture for the tool: Antoine Colin, Jean-François Deverge and Matthieu Avila.

References

1 Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 266–275, 2014.

2 aiT. `https://www.absint.com/ait/`.

3 Adnan Bouakaz, Isabelle Puaut, and Erven Rohou. Predictable binary code cache: A first step towards reconciling predictability and just-in-time compilation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 223–232, 2011.

4 Bound-T. `http://www.bound-t.com/`.

5 Chronos. `http://www.comp.nus.edu.sg/~rpembed/chronos/`.

6 Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2/3):249–274, May 2000.

7 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

8 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 68–77, 2009.

9 Damien Hardy and Isabelle Puaut. Predictable code and data paging for real time systems. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 266–275, 2008.

10 Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, December 2008*, pages 456–466, 2008.

11 Damien Hardy and Isabelle Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 57(7):677–694, 2011. Special Issue on Worst-Case Execution-Time Analysis. `doi:10.1016/j.sysarc.2010.08.007`.

12 Damien Hardy and Isabelle Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51(2):128–152, 2015.

13 B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011. `doi:10.1109/RTAS.2011.27`.

14 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2009. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2009.2283`.

15 Benjamin Lesage, Isabelle Puaut, and André Seznec. PRETI: partitioned real-time shared cache for mixed-criticality real-time systems. In *20th International Conference on Real-Time and Network Systems, RTNS'12, Pont a Mousson, France – November 8-9, 2012*, pages 171–180, 2012.

16 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 217–226, 2015.

17 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Richard Gerber and Thomas Marlowe, editors, *LCTES'95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, volume 30, pages 88–98, New York, NY, USA, 1995. doi:10.1145/216636.216666.

18 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

19 José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 497–502, 2012.

20 OTAWA. http://www.otawa.fr/.

21 Dumitru Potop-Butucaru and Isabelle Puaut. Integrated worst-case execution time estimation of multicore applications. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–31. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/OASIcs.WCET.2013.21.

22 Syed Aftab Rashid, Geoffrey Nelissen, Damien Hardy, Benny Akesson, Isabelle Puaut, and Eduardo Tovar. Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 262–272, 2016.

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

24 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil 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. *Journal of Systems Architecture*, 61(9):449–471, 2015. URL: http://www.jopdesign.com/doc/t-crest-jnl.pdf, doi:10.1016/j.sysarc.2015.04.002.

25 Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, November 1996.

26 SWEET. http://www.mrtc.mdh.se/projects/wcet/sweet/.

27 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.

28 Tutor 2016. https://tutor2016.inria.fr/.

29 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 Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. doi:10.1145/1347375.1347389.

The W-SEPT Project: Towards Semantic-Aware WCET Estimation

Claire Maiza[1], Pascal Raymond[2], Catherine Parent-Vigouroux[3], Armelle Bonenfant[4], Fabienne Carrier[5], Hugues Cassé[6], Philippe Cuenot[7], Denis Claraz[8], Nicolas Halbwachs[9], Erwan Jahier[10], Hanbing Li[11], Marianne De Michiel[12], Vincent Mussot[13], Isabelle Puaut[14], Christine Rochange[15], Erven Rohou[16], Jordy Ruiz[17], Pascal Sotin[18], and Wei-Tsun Sun[19]

1 **Grenoble Alps University, VERIMAG, Grenoble, France**
claire.maiza@univ-grenoble-alpes.fr
2 **Grenoble Alps University, VERIMAG, Grenoble, France**
pascal.raymond@univ-grenoble-alpes.fr
3 **Grenoble Alps University, VERIMAG, Grenoble, France**
catherine.parent-vigouroux@univ-grenoble-alpes.fr
4 **IRIT, University of Toulouse, Toulouse, France**
bonenfant@irit.fr
5 **Grenoble Alps University, VERIMAG, Grenoble, France**
fabienne.carrier@univ-grenoble-alpes.fr
6 **IRIT, University of Toulouse, Toulouse, France**
Hugues.Casse@irit.fr
7 **Continental AG, Toulouse, France**
philippe.cuenot@continental-corporation.com
8 **Continental AG, Toulouse, France**
denis.claraz@continental-corporation.com
9 **Grenoble Alps University, VERIMAG, Grenoble, France**
nicolas.halbwachs@univ-grenoble-alpes.fr
10 **Grenoble Alps University, VERIMAG, Grenoble, France**
erwan.jahier@univ-grenoble-alpes.fr
11 **Inria, IRISA, Rennes, France**
hanbing.li@inria.fr
12 **IRIT, University of Toulouse, Toulouse, France**
Marianne.De-Michiel@irit.fr
13 **IRIT, University of Toulouse, Toulouse, France**
mussot@irit.fr
14 **University of Rennes 1, IRISA, Rennes, France**
isabelle.puaut@irisa.fr
15 **IRIT, University of Toulouse, Toulouse, France**
rochange@irit.fr
16 **Inria, IRISA, Rennes, France**
Erven.Rohou@inria.fr
17 **IRIT, University of Toulouse, Toulouse, France**
Jordy.Ruiz@irit.fr
18 **IRIT, University of Toulouse, Toulouse, France**
Pascal.Sotin@irit.fr
19 **IRIT, University of Toulouse, Toulouse, France**
wsun@irit.fr

17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017).
Editor: Jan Reineke; Article No. 9; pp. 9:1–9:13

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

Abstract

Critical embedded systems are generally composed of repetitive tasks that must meet hard timing constraints, such as termination deadlines. Providing an upper bound of the worst-case execution time (WCET) of such tasks at design time is necessary to guarantee the correctness of the system. In static WCET analysis, a main source of over-approximation comes from the complexity of the modern hardware platforms: their timing behavior tends to become more unpredictable because of features like caches, pipeline, branch prediction, etc. Another source of over-approximation comes from the software itself: WCET analysis may consider potential worst-cases executions that are actually infeasible, because of the semantics of the program or because they correspond to unrealistic inputs. The W-SEPT project, for "WCET, Semantics, Precision and Traceability", has been carried out to study and exploit the influence of program semantics on the WCET estimation. This paper presents the results of this project : a semantic-aware WCET estimation workflow for high-level designed systems.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.2.2 Design Tools and Techniques, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, C.3 Special-Purpose and Application-Based Systems, B.4.4 Performance Analysis and Design Aids

Keywords and phrases Worst-case execution time analysis, Static analysis, Program analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2017.9

1 Introduction

Critical embedded systems are generally composed of repetitive tasks that must meet strong timing constraints, such as termination deadlines. Providing an upper bound of the worst-case execution time (WCET) of such tasks at design time is necessary to guarantee the correctness of the system.

Test based methods, widely used in practice, provide actual execution times but cannot guarantee that the worst case has been reached. Static analysis methods aim at providing a guaranteed upper bound to the WCET, by considering an abstract model of the program execution. In order to be safe, and also to keep the analysis tractable, the models are necessarily pessimistic and often lead to a possibly large over-approximation of the WCET.

In static WCET estimation, a main source of over-approximation comes from the complexity of the modern hardware platforms: their timing behavior tends to become more unpredictable because of features like caches, pipeline, test prediction, etc. Another source of over-approximation comes from the software itself: WCET analysis may consider potential worst-case executions that are actually infeasible, because of the semantics of the program or because they correspond to unrealistic inputs.

For instance, in the automotive application (Engine Management System : EMS) of Continental Corporation the modules of the application are mostly implementing generic algorithms that use calibration data for possible adaptation : a worst-case path could correspond to an unrealistic system state like high-engine speed with low-injection set point.

In the classical WCET estimation framework, the *data-flow analysis* is in charge of discovering infeasible execution paths. It must at least provide constant bounds for all the loops in the program, otherwise a finite WCET is not even guaranteed to exist. Apart from loop-bounds, control-flow analysis usually identifies simple semantics properties such as tests exclusions, that may prune infeasible execution paths when computing the WCET.

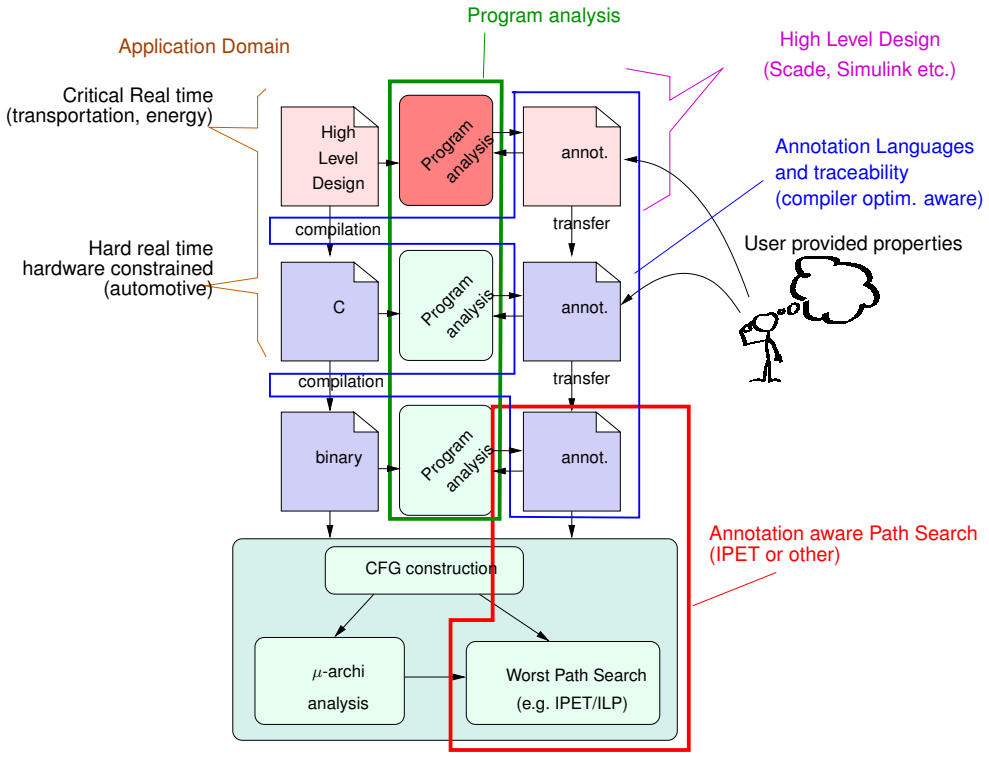


Figure 1 Work-flow and general organization of a semantic aware WCET estimation tool.

The W-SEPT project, for "*WCET, Semantics, Precision and Traceability*", has been carried out to study and exploit the influence of program semantics on the WCET estimation. This paper presents the results of this project: a semantic-aware WCET estimation workflow for high-level designed systems.

1.1 Workflow of a semantic-aware WCET estimation tool

The goal of the W-SEPT project[1] was to define and prototype a complete semantic-aware WCET estimation workflow [1]. It gathers researchers in the domain of timing and program analysis, together with an industrial partner from the real-time domain. The project mainly focuses on the semantic aspects, and thus, the pruning of infeasible paths. As far as possible, the idea is to extend and adapt the classical WCET estimation workflow. In particular, all that concerns the hardware analysis is inherited from previous work, through the use of the tool OTAWA[2].

This paper summarizes the main achievements of the project. We give the general picture: more details can be found in referenced papers. These achievements are structured according to the general workflow of the project, depicted by Figure 1.

It retains the general organization of classical existing tools [24]. The bottom block is the WCET computation tool itself, organized in three steps: Control-Flow graph (CFG) construction, micro-architecture analysis, and worst-path search on the CFG. Generally, this

[1] `http://wsept.inria.fr`
[2] `http://www.otawa.fr`

last step uses the classical Implicit Path Enumeration Technique (IPET) [14]. This WCET estimation takes as input the binary code of the program, and a set of semantic informations classically named *annotation file*, and containing at least the loop bounds.

These binary annotations come from program analysis. This analysis is generally performed at the source level, C language most of the time, rather than at the binary level. Indeed, analyzing C code is technically much simpler than analyzing binary code, but more importantly, the analysis often requires extra information that only the user can provide (e.g., inputs ranges, exclusion, implications). The user can probably express these properties in terms of the C variables, but it would be much harder or even impossible to do it in terms of the compiled binary code. This two-layers description raises the well-known problem of *traceability* of annotations when transferring information between layers.

So far, the principles depicted in Figure 1 are rather classical. An innovation of the project was to take into account a third layer in the design flow: the use of high-level design languages that are common in the domain of (critical) real-time applications. Classical examples of high-level design tools are Scade suite[3], used in avionics, energy or transportation, and Simulink/Stateflow[4] widely used in control engineering systems. These high-level design tools provide automatic code generation to C, which is no longer the source code, but only an intermediate code. A consequence is that user annotations and program analysis can be expressed and performed at the design level. The coupling of timing analysis and high level design is not new in itself. For instance the tool aIt (from the Absint company) has been coupled with the Scade Tool Suite[5]. However, this integration does not consider the extraction and exploitation of properties at the Scade level for enhancing the analysis of aIt.

The project proposed to focus on three main issues depicted by enclosing boxes in Figure 1:

- Program analysis, that can be performed at high-level design, C or binary level, and may take into account information provided by the user.
- Annotations and traceability between the language levels, which strongly involve the compilers: as far as possible, the compilation process should be annotation-aware, in the sense that the program transformations performed by the compiler should be reflected as annotation transformations.
- WCET estimation tool and the worst-path search step, must be adapted to take into account the richer kind of annotations produced by the workflow.

In this summary, we briefly describe the obtained results concerning each step of this workflow. In Section 2, we present how, at each stage, we can generate properties (automatically extracted) in order to discard infeasible paths. Then we show how to annotate these properties and automatically translate them through the compilation process, in Section 3. In Section 4, we describe how an existing WCET estimation tool was adapted in order to exploit this new kind of annotations. Even enhanced thanks to semantic information, the WCET estimation is still necessarily a pessimistic upper bound. Section 5 presents two methods for assessing the pessimism, by finding a guaranteed (big) lower bound, and thus an interval containing the actual WCET. Indeed, the smaller is the interval, the better is the estimation.

[3] `http://www.esterel-technologies.com/products/scade-suite`
[4] `http://mathworks.com/products/simulink/`
[5] `http://www.absint.com/ait/scade.htm`

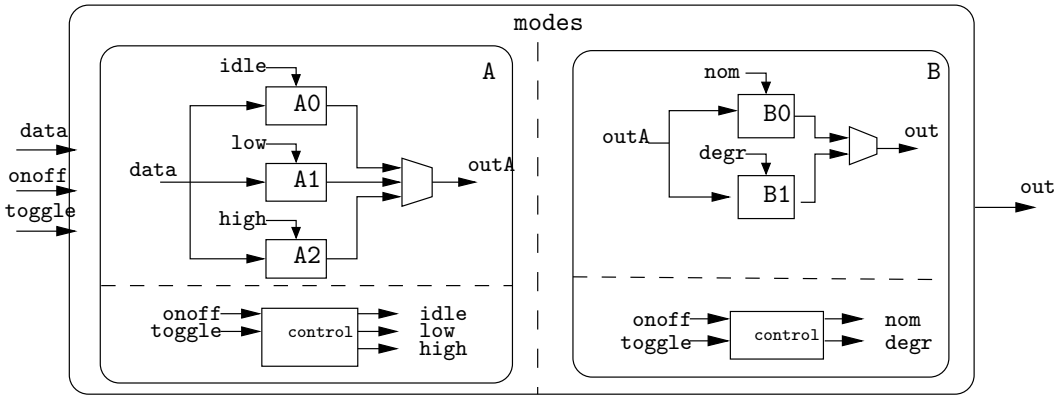


Figure 2 A typical high-level dataflow design.

2 Extraction of semantic properties

In this section, we explain what kind of semantic properties may help to enhance the WCET estimation: where do they come from and which step of the application development do they refer to (binary, code, design). We consider the automatic extraction of properties. The set of analyses should lead to a cumulative improvement as the kind of properties they cover should be exclusive. The annotation language, necessary to express and transfer user assumptions and discovered properties, is presented in the next section.

2.1 High-level properties

Critical embedded systems are often designed using high level modeling languages, such as Scade or Simulink. The system is then automatically compiled into classical imperative code (C in general), and then into binary code (cf. Fig 1).

Figure 2 shows a typical high-level data-flow design. For simplicity, it is represented as a diagram, while the actual program is written in Lustre [7], the academic textual language which is the ancestor of the industrial Scade language. This application consists of two sub modules, `A` and `B`, each of them consisting in two parts: a control part and a data processing part. The data processing part has different computation modes (e.g., `A0`, `A1` and `A2`), controlled by a *clock* (e.g., `idle`, `low` and `high`). An important property of such a design is that these modes are exclusive: at each reaction exactly one of the modes is activated. This information, obvious at the design level, may or may not be obvious at the C or binary level: depending on the compilation process, the (high level) mode exclusion may result or not into structurally exclusive pieces of code. In a more subtle way, we also know, for this particular program, that there is a logical exclusion between the modes of the two sub-modules: if `A` is not `idle` (`A1` or `A2`), then `B` is necessarily in degraded mode (`B1`). This property is neither structural nor obvious: it is an *invariant* of the infinite cyclic behavior of the application that holds if we suppose `toggle` and `onoff` are never true at the same time (which is an hypothesis on the system environment). It is therefore almost impossible to discover it at the low-level.

Based on these remarks, we have developed a prototype for discovering such properties, propagate them through the compilation process, and exploit them to enhance the WCET estimation. Details on how these properties are transferred and used to enhance the WCET are in Section 3.2 and 4.1.

To extract properties that may reveal infeasible paths, we identified the high-level expression that influences a branch at binary level. In a second step, we use a model-checker (Lesar [18]) to check the validity of properties at the Lustre level. We use two heuristics:

- a pairwise algorithm: The high level code is analyzed to find a set of interesting control variables, according to a simple heuristic: any Boolean variable that controls computation modes (often called the *logical clocks*) is likely to control big pieces of binary code, and thus, has a big influence on the computation time. In the example, the five control variables are selected. We "blindly" search for all possible pairwise relations (either exclusions or implications) between these variables. For n variables, there are $4(n*(n-1)/2) = 2n(n-1)$ such (potential) relations (40 in the example). For each relation proven by Lesar, we generate the corresponding constraints at the binary level thanks to the traceability information; in the example, 5 over 40 relations are proven.
- an iterative algorithm: according to the traceability information, the validity of the worst-case path is translated (if possible) into a logical condition on the high-level variables (e.g. $\neg$`idle` $\wedge$ `low` $\wedge$ `nom`); Lesar is called to check this condition; if the condition is unsatisfiable, the WCET path candidate is proven unfeasible, the corresponding constraint gives an infeasible path that we give to the WCET estimation tool; we restart to find a new candidate, and so on. If the condition is found satisfiable, the process stops with the current WCET.

The improvement on the WCET estimation is important and similar for both strategies (up to 50% on a realistic Lustre benchmark). The iterative algorithm may be relatively costly. The pairwise strategy has a constant overhead. The whole experiment is presented in details in [20].

2.2 C level properties

The discovery of bounds and relations on numerical variables is a classical goal in program analysis [5]. These bounds and relations can obviously be used to restrict the set of feasible paths considered in WCET evaluation. This can be helped by adding some counters to the code of the program: of course, adding a loop counter may result in finding a bound to this counter, and thus to the iteration number. Moreover, adding block counters, and finding relations between these counters can reveal subtle restrictions in the possible executions of the program.

An analysis of this instrumented program with counters using an analyser of linear relations (here, we used the tool PAGAI [8]), automatically discovers some linear relations on counters. This approach has been implemented in a prototype tool [2], and applied in combination with OTAWA to several examples. Results show improvements of the evaluated WCET (with or without counters) up to 50% on TACLeBenchs[6].

2.3 Low-level properties

Looking for infeasible paths at binary level benefits from the exact matching of the program with the hardware, and to inject found properties immediately in the WCET computation. The price is an increase of the analysis time caused by the program size and the loss of expressivity implied by machine instructions. Consequently, existing analyses either look for very simple infeasible paths [6, 22], or design a new WCET computation method [22]. Our

[6] `http://www.tacle.eu/index.php/activities/taclebench`

approach tries to get rid of these limitations by using SMT solvers (Satisfiability Modulo Theories) to generate infeasible path properties [21]. This approach finds a large set of infeasible paths on the TACLeBenchs: it cuts from 1 to some thousands of edges in the control flow graph.

2.4 Delta-guided extraction

In order to lower the real WCET, some approaches compute execution time profiling (using the estimation of program part execution time with respects to the global WCET) [3] or generate a static profile using probabilities for decisions at branching points [25]. The delta tool [27] aims at identifying the conditional statements that are unbalanced in terms of execution time weight (obtained so far by a naive counting of instructions). This highlights, to the user or the program analyzers, the parts of code where a semantic analysis or user annotation should focus to gain more accuracy on the WCET estimation. Branching statement analysis allows identifying parameters as important or not due to their unbalanced weight.

This method may be combined with any of the extraction method presented in this section. For instance, it may help reducing the number of pairwise properties to check at the high-level: if the validity of a pairwise property does not influence the WCET, there is no need to call the model-checker.

3 Annotation language and traceability

In order to express most of the properties, we use and extend the existing FFX annotation language [26]. FFX is an open, portable, and expandable annotation format. It allows combining flow fact information from different high-level tools. It is used as an intermediate format for WCET analysis; in particular it is both the source and target language for the traceability tools, that transfer information from one level to the next one.

3.1 Annotation language

The principle of FFX is to express a wide class of information that may be helpful to compute or enhance the WCET estimation by Otawa. It is specifically dedicated to sequential programs (C or binary), and allows expressing both data-flow and control-flow properties.

- Data-flow: FFX allows one to identify data and express properties such as the type, the range, the mutability status (local, global, input or output); such information is given "as is" to Otawa and, may (or may not) be used for the computation of the WCET.
- Control-flow: FFX allows one to identify control points and express constraints and relations between them. Control points are typically identified by line numbers in C code, or with address offsets in binary code. Classical flow information concerns the maximum occurence number of a program point (loop bound), the fact that a branch is always or never taken etc.

The FFX language and associated tools have been extended and adapted to meet the goals of the project:

- Transfer of ILP constraints: the principle of control point counters and constraints, already existing for expressing the loop bounds has been extended to any kind of linear relations between counters. This way, flow information discovered by analyzing tools (cf. Section 2) can be directly transferred to the worst-path analysis module.
- Logical paths constraints: the language has been extended to express path properties in a more *logical* way; the exclusion between several control points within a particular

scope. This kind of information can be translated later into classical ILP constraints, or be handled by the new concept of *Path Property Automata* (PPA), as presented in the next Section 4.

- Expression of specific scenarios: a strong requirement from the industrial partner is to be able to evaluate the WCET under some specific use cases. This notion is different from the classical constraints, since scenarios can contradict each other, and a single FFX file may contain several scenarios. Since FFX is not designed to be used by humans, a mini language of user annotations has been designed to be used directly as "pragmas" in the C code. These source-code annotations are extracted and automatically translated in FFX.

3.2 Traceability

From design level to source code, we transfer the properties by tracing them in the code generator (by inserting additional comments in the C code).

From C to binary, hundreds of compiler optimizations may have a strong impact on the structure of the code, making it impossible to match source-level and binary-level control flow graphs. This ends up in a loss of useful information. For this reason, the current practice is to turn off compiler optimizations, resulting in low average-case and worst-case performance. To safely benefit from optimizations (as in [11]), we propose a framework to trace and maintain flow information up-to-date from source code to machine code [12].

The transformation framework, for each compiler optimization, defines a set of formulas, that rewrite available semantic properties into new properties depending on the semantics of the concerned optimization. Supported semantic properties are *loop bounds* and linear inequations constraining the execution counts of basic blocks. Consider, for example, loop unrolling, that replicates a loop body k times to reduce loop branching overhead and increase instruction level parallelism. The associated rewriting rule divides the initial loop bound by k, and introduces constraints on the execution counts of the basic blocks within the loop (see [12, 13] for details).

We have implemented this traceability in the LLVM compiler infrastructure (local patches). Each LLVM optimization was modified to implement the rewriting rules corresponding to the optimization. Semantic information is initially read from a file in the FFX format, and then represented internally in the LLVM compiler, and finally transformed jointly with the code transformations. Note that, if a transformation happens to be too complex to trace the information, it can be disabled. This is a better situation than the general current practice which is disabling all optimizations.

4 Exploitation of semantic properties in WCET estimation

Sections 2 and 3 respectively presented how properties are extracted, expressed and traced. This section presents how the properties are taken into account in the WCET analysis.

There are basically two ways for handling infeasible path properties in WCET analysis. The explicit way proceeds by control-flow graph transformations and aims at pruning (any) infeasible paths from the model. This general method is virtually able to handle any kind of pruning properties, but may lead to the explosion of the model size. The implicit way, as formalized in the classical IPET/IPET method, prevents the model size explosion by summarizing "big" families of infeasible paths with "small" numerical relations. In the project, we have first considered the implicit way, and then proposed a more versatile method allowing a mix between explicit and implicit exploitation

4.1 Exploitation in ILP

In many cases, the properties found at the C or high level can be expressed relatively directly into ILP constraints, exploitable in the classical IPET framework.

This is typically the case for *branch conflicts* properties: a conflict is a set of control points in the program that cannot be all taken during the same execution. Expressing simple conflict patterns in ILP is rather classical, and numerous examples can be found in literature. For instance, a conflict between three control points a, b, c, within a loop bounded by the constant n, can be expressed with: $a + b + c \leq 2 \times n$.

We have presented in [19] a method that generalizes the encoding of conflicts with ILP constraints. In contrast with many existing approaches, it is not based on patterns, but on the ability of counting how many times a conflict occurs. Thanks to this principle, one can handle for instance conflicts occurring between different loop scopes. Consider for instance the following program structure, with two nested loops:

```
for i = 1 to n do
   if ... then a else ...
   for j = 1 to m do
      if .. then b else ...
```

and suppose that some analysis has established that, if a is taken during one outer iteration, b cannot be taken during the whole forthcoming inner loop. Our method gives that this particular conflict can be expressed with the following ILP constraint:

$$m \cdot a + b \leq n \cdot m\,.$$

The main advantage of the ILP encoding is that the results of semantic analysis can be directly exploited into existing WCET tools based on the Implicit Path Enumeration Technique. However, even if the ILP encoding is safe (only infeasible paths are pruned), it is not necessarily exact (it may still accept some infeasible paths).

4.2 Exploitation through automata

We propose a general, versatile, and non-intrusive process for the integration of the paths properties [15, 16, 17]. This process assumes that the WCET tool internally handles CFGs and integer linear constraints, which is the case of every IPET-based WCET analyzers. The internal representation of the program is extracted, improved according to the annotations and set back in the tool. The transformation relies on a novel automata-based formalism that can represent both the program CFG and the annotations. The transformation itself is defined as an automata product; it results is an automaton whose paths are both existing in the original CFG and valid with respect to the annotations[7].

Figure 3 shows two Path Property Automata (PPA). On the left, a PPA isomorphic to a program CFG. On the right, a corresponding PPA that additionally reflects the property "in each iteration of the loop starting with E and ending with X, at most one of A or B can be taken".

Note that our formalism mixes explicit (state/transition) and implicit (local counter α, bounded by one) concerns, and can be exploited accordingly. The purely explicit exploitation consists in producing a flat product of the graph and the property automaton. In this case,

[7] http://www.mrtc.mdh.se/projects/WTC/

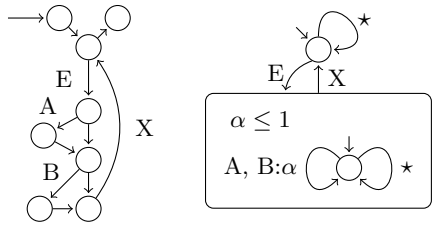


Figure 3 Path Property Automaton.

the resulting modified control graph is a graph where the core of the iteration is replaced by 3 exclusive paths: (1) A is executed and B is not, (2) A is not executed and B is, (3) none of them are executed.

In order to limit the size explosion, we have also defined a specific product that keeps, as far as possible, the counter constraints implicit. In this particular example, the algorithm results on an unchanged control graph decorated with constraints: $\alpha = A + B \quad \alpha \leq E$, which are equivalent to the classical conflict constraint $A + B \leq X$.

The analysis performed on the enriched CFG delivers a WCET improvement up to 10% on the benchmarks of the WCET Tool Challenge.

5 Assessment of WCET estimations

The approaches presented in this paper aim at increasing the precision of WCET estimations by enhancing the knowledge of possible execution paths. A reduction on the estimated WCET reflects a tighter analysis due to taking the program semantics into consideration. However, whether the new estimation is far from or close to the real WCET remains unknown. In order to evaluate the precision of static WCET analysis, two approaches have been studied. The first approach is based on simulation, and aims to assess the precision by comparing the estimated WCET with the longest observed execution time. The second approach is integrated to the estimation process, and aims at self-assessing the pessimism of the static analysis.

5.1 Simulation-based assessment

We have developed a simulator (OSIM [23]) that uses the same hardware model as OTAWA, in order to provide execution times that are consistent with those provided by the static analysis. Therefore, for a given program, the simulation provides a *guaranteed lower bound*, M_{WCET}, that forms with the guaranteed upper bound E_{WCET} an uncertainty interval. The relative size of this interval can be measured by an *over-estimation ratio*: ρ=(E_{WCET}- M_{WCET}) / M_{WCET}. The first way to reduce ρ is to decrease E_{WCET} with more precise static analysis. The second way is to increase the $M_{WCET}$with more thorough test generation.

In the case of reactive programs, which continuously interact with their environment, performing intensive test generation requires to simulate environments that can react to the system outputs. This is achieved via the use of LUTIN [9, 10], a language for specifying and playing random constrained reactive scenarios.

The scenario written for simulation must at least integrate the same hypothesis as the static analysis. If it is not the case, M_{WCET} and E_{WCET} are considering different sets of realistic executions, and their results are not inconsistent. Consider for instance, the example

in Section 2.1: the estimation E_{WCET} is obtained under the assumption that the inputs `toggle` and `onoff` are exclusive. When simulating the program without this assumption we obtain a M_{WCET} which is 60% greater than the E_{WCET} [23].

Writing in Lutin a random scenario, that takes only input constrains into account, requires very little effort. In many cases, such a simple scenario gives good results. For instance, for the example of Section 2.1, we have obtained this way a ratio ρ of about 5%.

In some cases, simple random testing is not sufficient to get close to the worst case, and the user expertise is necessary to write more sophisticated scenarios, that drive the generation to uncommon and costly executions.

5.2 Quantification of static analysis pessimism

The simulation-based approach provides a safe upper bound on the pessimism of the estimated WCET, but it does not give any insight into the sources of pessimism: it could be due to highly dynamic hardware schemes, the behavior of which must be approximated at analysis time, or to under-specified flow information.

In [4], we introduced a framework that extends static WCET analysis to quantify the possible overestimation. The approach consists in identifying, during the analysis, whether the intermediate timing information is certain or uncertain. This identification is done when over-approximation is necessary (e.g., when merging abstract program states). This way, two WCETs estimations are computed: one that is a classical and pessimistic upper bound of the real WCET, and one that results from program/hardware states that are known to be reachable.

6 Conclusion and future work

In this paper, we summarized the semantic-aware WCET estimation workflow proposed in the W-SEPT project. From semantic properties extracted at high level, C level or binary level by static analysis or user annotation, we express them with the FFX annotation language, and trace them down to the binary level. The WCET estimation tool OTAWA has been adapted to integrate those properties through CFG modifications or ILP-based constraints. Generally, the results are good and show that the semantic-aware WCET workflow is a good opportunity to gain precision in WCET estimation. Moreover, we propose two methods for assessing the estimated WCET, by computing an *over-estimation* ratio that measures its (possible) relative pessimism; one method is based on constrained random simulation, while the other proceeds directly during the analysis.

This W-SEPT project highlighted that in the context of reactive systems, the semantic-aware WCET analysis may largely gain precision. In a future project, we aim at focusing on the specific context of reactive systems and synchronous languages, and consider the relations between timing analysis and certified code generation (e.g., DO 178C in avionics)

References

1 Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Hanbing Li, Claire Maiza, Marianne De Michiel, Vincent Mussot, Catherine Parent-Vigouroux, Isabelle Puaut, Pascal Raymond, Erven Rohou, and Pascal Sotin. When the worst-case execution time estimation gains from the application semantics. In *8th European Congress on Embedded Real-Time Software and Systems (ERTS2 2016)*, Toulouse, France, 2016.

2 Remy Boutonnet and Mihail Asavoae. The WCET analysis using counters – a preliminary assessment. In *8th JRWRTC, in conjunction with RTNS*, Versailles, France, 2014.

3 Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In *20th International Conference on Real-Time and Network Systems (RTNS 12)*, Pont à Mousson, France, 2012.

4 Hugues Cassé, Haluk Ozaktas, and Christine Rochange. A framework to quantify the overestimations of static wcet analysis. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OASIcs*, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.1.

5 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, Tucson, Arizona, January 1978.

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

7 Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

8 Julien Henry, David Monniaux, and Matthieu Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.

9 Erwan Jahier, Simplice Djoko-Djoko, Chaouki Maiza, and Eric Lafont. Environment-model based testing of control systems: Case studies. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, Grenoble, France, 2014.

10 Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Engineering functional requirements of reactive systems using synchronous languages. In *International Symposium on Industrial Embedded Systems (SIES 2013)*, 2013.

11 Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Journal on Real-Time Systems*, 45(1-2), 2010.

12 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *22nd International Conference on Real-Time Networks and Systems (RTNS'14)*, Versailles, France, 2014.

13 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter wcet estimation: Application to vectorization. In *21st IEEE International Conference on Embedded Systems and Real-Time Computing Systems and Applications, RTCSA'15*, Hong Kong, China, 2015.

14 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

15 Vincent Mussot, Armelle Bonenfant, Pascal Sotin, Philippe Cuenot, and Denis Claraz. From relevant high-level properties to WCET computation improvement. In *International Conference on Embedded Real Time Software and Systems (ERTS2 2013)*, Toulouse, France, 2013.

16 Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne de Michiel, and Hugues Cassé. Expressing and exploiting path conflicts in wcet analysis. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OASIcs*, pages 3:1–3:11. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.3.

17 Vincent Mussot and Pascal Sotin. Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded Systems and Real-Time Computing Systems and Applications, RTCSA'15*, Hong Kong, China, 2015.

18 Pascal Raymond. Synchronous program verification with lustre/lesar. In S. Mertz and N. Navet, editors, *Modeling and Verification of Real-Time Systems*, chapter 6. ISTE/Wiley, 2008.

19 Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *International Conference on Embedded Software (EMSOFT 2014)*, New Dehli, India, 2014.

20 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. Timing analysis enhancement for synchronous program. *Real-Time Systems*, pages 1–29, 2015. doi:10.1007/s11241-015-9219-y.

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

22 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *43rd annual Design Automation Conference (DAC'06)*, San Francisco, California, 2006.

23 Wei-Tsun SUN. A framework for simulate synchronous reactive programs and measure execution times to aid wcet analysis. Technical Report 27-06-2016, Verimag Research Report, 2016.

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

25 Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *27th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO94)*, San Jose, California, 1994.

26 Jakob Zwirchmayr, Armelle Bonenfant, Marianne de Michiel, Hugues Cassé, Laura Kovács, and Jens Knoop. FFX: A portable WCET annotation language (regular paper). In *20th International Conference on Real-Time and Network Systems (RTNS 12)*, Pont à Mousson, France, 2012.

27 Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot. Identifying relevant parameters to improve WCET analysis. In *14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, volume 39 of *OASIcs*, pages 93–102. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/OASIcs.WCET.2014.93.

The P-SOCRATES Timing Analysis Methodology for Parallel Real-Time Applications Deployed on Many-Core Platforms*

Vincent Nelis[1], Patrick Meumeu Yomsi[2], and Luís Miguel Pinho[3]

1 **CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Porto, Portugal**
`nelis@isep.ipp.pt`
2 **CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Porto, Portugal**
`pamyo@isep.ipp.pt`
3 **CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Porto, Portugal**
`lmp@isep.ipp.pt`

Abstract

This paper presents the timing analysis methodology developed in the European project P-SOCRATES (Parallel Software Framework for Time-Critical Many-core Systems). This timing analysis methodology is defined for parallel applications that must satisfy both performance and real-time requirements and are executed on modern many-core processor architectures. We discuss the motivation and objectives of the project, the timing analysis flow that we proposed, the tool that has been developed to automatize it, and finally we report on some of the preliminary results that we have obtained when applying this methodology to the three application use-cases of the project.

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

Keywords and phrases Timing analysis, many-core platform

Digital Object Identifier 10.4230/OASIcs.WCET.2017.10

1 Introduction and context of the work

The objective of the European project P-SOCRATES [7] (Parallel Software Framework for Time-Critical Many-core Systems) was to develop a new design framework to allow current and future applications with high-performance and real-time requirements to fully exploit the huge performance opportunities brought by the most advanced many-core processors, whilst ensuring a predictable performance and maintaining (or even reducing) the development costs of the applications. The main outcome of the project is the UpScale SDK [1].

Upscale is a framework for the development of real-time high-performance applications in many-core platforms. The SDK targets systems that demand more and more computational performance to process large amounts of data from multiple data sources, whilst

* This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement no. 611016 (P-SOCRATES).

requiring guarantees on processing response times. Many-core processor architectures allow these performance requirements to be achieved, by integrating dozens or hundreds of cores, interconnected with complex networks on chip, paving the way for parallel computing. Unfortunately, parallelization brings many challenges, by drastically affecting the system's timing behavior: providing guarantees becomes harder, because the behavior of the system running on a multi-core processor depends on interactions that are usually not known by the system designer. This causes system analysts to struggle to provide timing guarantees for such platforms. UpScale tackles this challenge by including technologies from different computing segments to successfully exploit the performance opportunities brought by parallel programming models used in the high-performance domain and timing analysis from the embedded real-time domain, for the newest many-core embedded processors available.

Most of the current state-of-the-art software techniques for analysis and scheduling assume that the system activities (the tasks) are functionally independent and most of their parameters, including their worst-case execution time (WCET), are known at design time. However, at run-time, the tasks that are co-scheduled on different cores share hardware resources, including caches, communication buses and main memory. Those resources introduce implicit functional dependencies among the tasks, as concurrent accesses to the same resource are not allowed, affecting their timing behaviour. This effect is exacerbated when scaling to many-core architectures. Therefore, current analysis and scheduling techniques cannot be applied as-is and need to be augmented to include all the sources of contention due to the increased number of shared resources.

As part of P-SOCRATES, we have developed a new timing analysis methodology to estimate the maximum execution time of parallel applications running on a many-core architecture. Preliminary results towards this direction have already been presented (see for example [4, 3, 5, 9, 8]) but we will not elaborate on those works in this paper. Our timing analysis methodology has been automated and the corresponding tool is now part of the UpScale SDK. In this paper, we describe the methodology, its automation tool, and we discuss briefly the results obtained by running it on the three project use-cases on the Kalray MPPA-256 many-core development board [2].

2 Overview of the application structure and execution model

In the P-SOCRATES execution model, the real-time applications start their execution on the host cores of the accelerator (i.e. the IO cores in the Kalray MPPA). To execute faster, they can offload some parts of their computation onto the accelerator at any time during their execution. The offloaded parts are organized in sets of potentially parallel segments of code that may be modeled as a directed acyclic graph (DAG). DAG nodes are implemented with OpenMP tasks with edges representing dependencies among these tasks (implemented with the *depend* OpenMP clause). We call each offloaded part, a "phase" of the application. Each phase can thus be seen as a graph of openMP tasks that can execute concurrently on different clusters. The focus of our analysis work is on the DAGs, where little previous works exist. Traditional real-time techniques can then be used to integrate the full computation, i.e. the sequential execution on the host cores and the offloaded phases.

To analyze the timing behavior of the applications, we define two different analysis flows: one for systems that use dynamic mapping and scheduling techniques and another one for static techniques. Due to space limitations, we only provide a brief summary of the differences between these two approaches. When *dynamic* techniques are used, the task-to-thread mapping of the lightweight runtime is performed during execution, based on

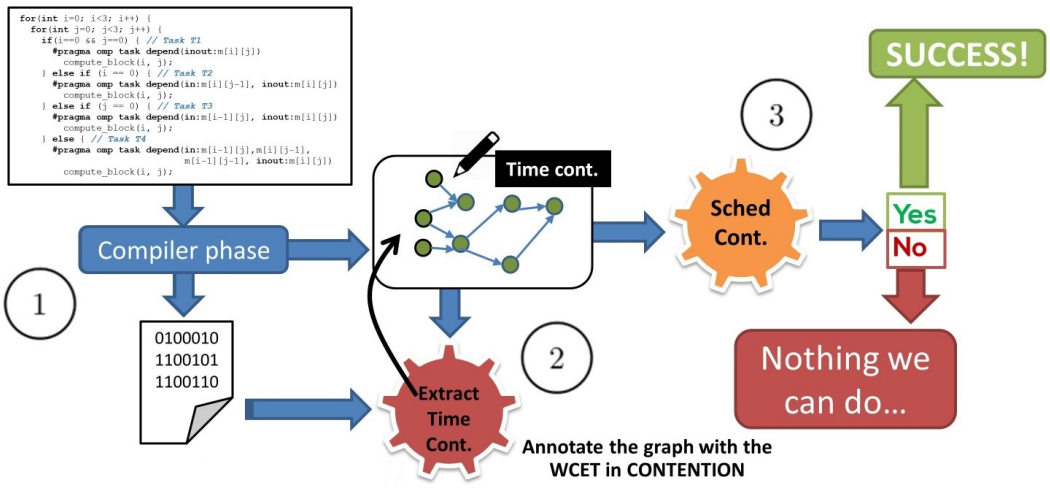


Figure 1 P-SOCRATES analysis flow using dynamic mapping and scheduling.

OpenMP task mapping algorithms, and the scheduling in the operating system uses global migration of threads. Therefore the system adapts better to variability in the load, achieving in principle higher performance (due to better load balancing). However, as variability is higher, less information exists to perform both timing and schedulability analyses, which may lead to pessimistic theoretical worst-case scenarios (which in practice may not occur). When *static* approaches are used, a specific mapping algorithm assigns the OpenMP tasks to particular threads in the OpenMP runtime, with each thread then statically assigned to a particular core, with the operating system using partitioned per-core schedulers. In this case, the system is less adaptable, in many cases incapable of taking advantage of eventual spare capacity (a core may be idle while tasks wait in other cores), which may lead to worse average performance. However, there is more information on the configuration of the system, which may allow a tighter analysis, leading to improved worst-case scenarios.

3 Analysis flow for systems using dynamic mapping and scheduling techniques

Figure 1 depicts the end-to-end analysis flow. As mentioned above, adopting the dynamic scheduling scheme means that all the OpenMP tasks are not pinned to a specific thread and can migrate from one core to another at runtime. In this case, there is only one queue for all pending/waiting OpenMP tasks. As soon as a core finishes the execution of a task, it starts (or resumes) the execution of the first task in that single shared queue. To analyze this configuration, the implemented steps are as follows.

Step ①. The application is compiled and the compilation of its source code generates at least two files: a multibinary file – the executable file to be run on the MPPA – and one DAG per application phase. Every phase (i.e., every "#pragma omp target" directive found in the source code that offload parts of the computation to the MPPA accelerator) leads to the creation of a set of omp tasks generated and executed in the accelerator. This set of tasks and their inter-dependencies are represented by a Task Dependency Graph (TDG) created at this point by the compiler. More precisely, the TDG represent the graph of dependencies between all the tasks.

Step ②. In this step, we estimate the maximum execution time of every openMP task when it suffers a maximum interference on the shared resources. We call this execution time the MEET of the tasks (Maximum Extrinsic Execution Time). The MEET is thus measured by enforcing the "nastiest" execution environment in which the analyzed tasks suffer as much interference as possible from other tasks and applications running concurrently. In these execution conditions, every analyzed task is executed multiple times over the same set of "worst" input data. This step requires to slightly modify the source code of the application and insert specialized code that will artificially generate those nasty execution conditions at runtime. In short, in this step all the openMP tasks of the application execute sequentially on the same core, by a single thread, while all the other threads execute on the other cores and generate as much interference as possible on the shared resources (memory, NoC, etc.). At the end of this step, every node of every TDG (which represents an OpenMP task) is annotated with its MEET.

Step ③. The annotated TDGs and the timing parameters of the applications (period and deadline) are given as input to the schedulability analysis tool to check whether all the timing requirements are met (i.e., no deadline is missed). Since in the case of dynamic mapping and scheduling we do not know beforehand on which core the tasks will execute, nor concurrently with which other tasks, the schedulability analysis must use the MEET as the worst-case execution time of every task, i.e., it assumes a maximum interference on every task. If the schedulability analysis fails, we have no other option than changing the timing parameters of the application (period and deadline) or optimizing its source code. We will see in the next section that we can reduce this pessimism by using a static scheduler.

4 Analysis flow for systems using static mapping and scheduling techniques

By using a static scheduling scheme, more information is available at design time. For example, the task-to-thread and thus the task-to-core mappings are known and, for every application, the subset of openMP tasks running concurrently is also known. This additional information allows us to considerably reduce the inherent pessimism of a dynamic scheduler, using a more complex analysis flow (see Figure 2).

Steps ❶. The first step is similar to Steps ① and ② for dynamic scheduling. The MEET is computed for every OpenMP task of the TDGs and is annotated to the graphs.

Steps ❷. The mapping tool is used to define a static task-to-core mapping that is schedulable. If the tool finds a valid mapping then the process ends and returns it. Note that the actual execution time of the tasks at runtime can only be lower than their MEET since the runtime interference will be lower. This means that any mapping found at this step that meets all the deadlines while assuming the MEET of every task is also guaranteed to meet all the deadlines for shorter tasks execution times.

Step ❸. If the mapping tool does not find any mapping that meets all the timing requirements, then we check if there exists a feasible mapping in the opposite scenario, i.e. while assuming the best execution conditions. That is, we measure in this step the maximum execution time of every openMP task when there is no interference whatsoever on the shared resources. To do so, similarly to the extraction of the MEET, we run all the openMP tasks

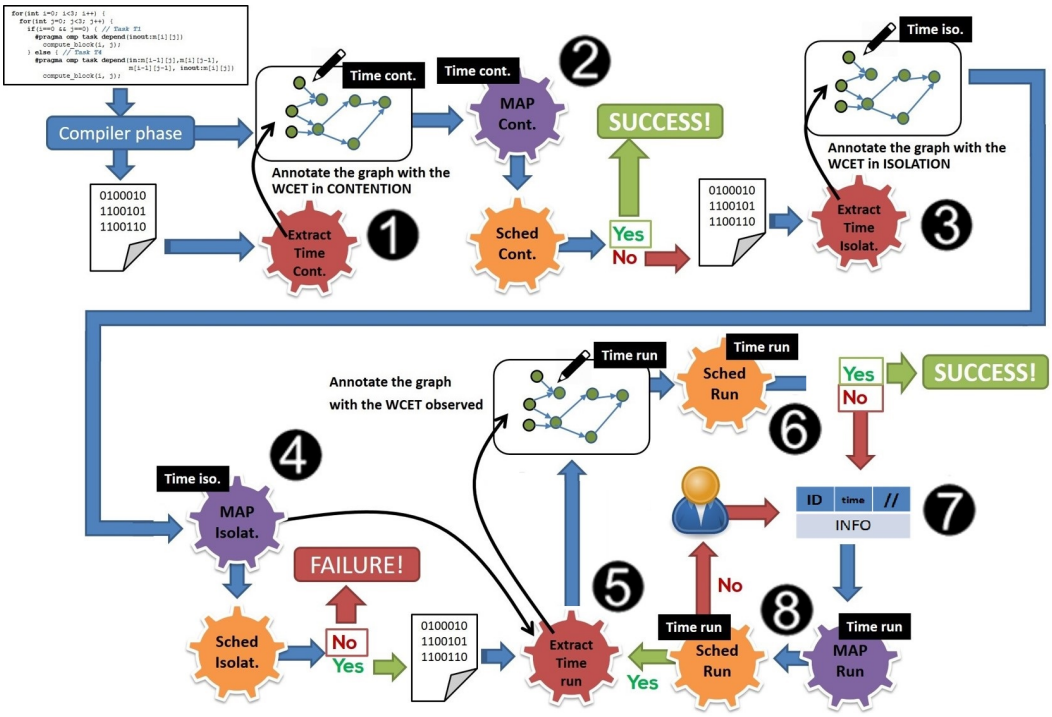


Figure 2 P-SOCRATES analysis flow using static mapping and scheduling.

sequentially on one core, by a single thread, but this time all the threads running on the other cores stay idle (busy-waiting) and do not use any resource. We call this maximum execution time the "MIET" of the tasks (Maximum Intrinsic Execution Time). At the end of this step, the MIET of every OpenMP task is annotated to the TDGs. The reason for measuring the MIET of the tasks is because in the case of static scheduling, we know that some tasks will not execute concurrently with some others (if they are mapped to the same core for example) and thus assuming the MEET was the most pessimistic assumption we could make. The MIET, on the other hand, is the most optimistic one.

Steps ❹. This step is similar to step ❷. The mapping tool tries to derive a mapping that meets all the timing requirements but contrary to step ❷, it assumes that all the OpenMP tasks execute for their MIET (instead of using the MEET estimates as in step ❷). If it does not find a feasible mapping then the process stops. In this case, there is no way to find a feasible mapping since the tool did not even find one assuming the best execution conditions, i.e. no interference whatsoever between the OpenMP tasks. The application is thus not schedulable. If the tool did find a valid mapping, then we need to check whether that mapping still meets all the timing requirements even with the interference that will occur at runtime, which is the next step.

Step ❺. This step runs the application by using the static mapping found at step ❹ and records the maximum actual execution time of every OpenMP task (also referred to as MAET – Maximum Actual Execution Time). Those MAETs are also added to the TDGs information.

Step ❻. The schedulability analysis tool is now used to check if the mapping that has been tested at the previous step is still schedulable when assuming for each OpenMP task its maximum actual/observed execution time (MAET) recorded at Step ❺. If this is the case, then the process stops and returns the mapping. Otherwise, the process moves on to Step ❼.

Step ❼. At this stage, we know that the mapping defined at step ❹ meets all the timing requirements if the tasks do not execute for longer than their MIET. However, when executed at step ❺, it turned out that some of the tasks took longer to complete (as potentially expected) and with those new estimations of their execution times (i.e., the MAETs), the mapping failed the test at step ❻. In this case, we display a table containing for each task: its MIET, MAET, MEET, and the set of tasks it has potentially been executed concurrently with (i.e., the subset of tasks allocated to a different core). If a task has a MAET closer to its MEET than to its MIET, we can conclude that the execution of that task is strongly impacted by the tasks that execute concurrently. Hence, it may be wise not to execute them on different cores. On the contrary, if its MAET is closer to its MIET, it means that the task is barely affected by the execution of the concurrent tasks and it may be judicious not to change that task-to-core allocation. During that step, we let the user register a "restriction", telling the mapper that in the next mapping to be produced a particular task cannot be assigned to the same core as the set of tasks it is currently mapped with. Those restrictions will be taken into account by the mapping tool at the next step ❽ when trying to find another mapping.

Steps ❽. The mapping tool is used again to derive a new mapping. This time the tool considers the restrictions defined by the user at step ❼ and assumes the tasks execution times obtained at step ❺, i.e. the MAETs. If the tool still does not find any valid mapping, then the user may try another set of restrictions with less, or different constraints. If this time a valid mapping is found, then we must re-test it, i.e., we must re-extract the maximum execution times (i.e., the MAETs) observed at runtime *for that particular mapping* and re-check the schedulability of the system while assuming those new MAETs. That is, the process goes back to Step ❺ with the mapping obtained at this step.

Except for these last two steps of defining task restrictions to guide the mapping, which requires manual intervention, the whole analysis process has been automatized within the analysis tool presented in the next section. The feedback loop could also be automatized, but there is yet no known heuristic for the mapping which is able to converge to a stable state.

5 Our P-SOCRATES timing analysis tool

Figure 3 is a screenshot of the tool developed to run the entire timing analysis methodology for a given application. The tool is written in Python 2.7 and is now part of the UpScale SDK, available at [1]. It offers a generic interface to easily implement, connect, and run together different application scripts, possibly written in different programming languages. The tool is based on three main concepts: commands, actions, and variables.

The **commands** are the basic blocks of the tool. A command is defined in a specific programming language and has a pre-defined type. A command is typically a small script that is used to perform a specific operation and its type describes how it must execute it. For example, one can define the commands "Upload the source code to the testing device" or "Compile the code remotely on that device". The former is of type "SFTP – Put" and its code lists the files to be uploaded (following a pre-defined syntax), whereas the latter can be

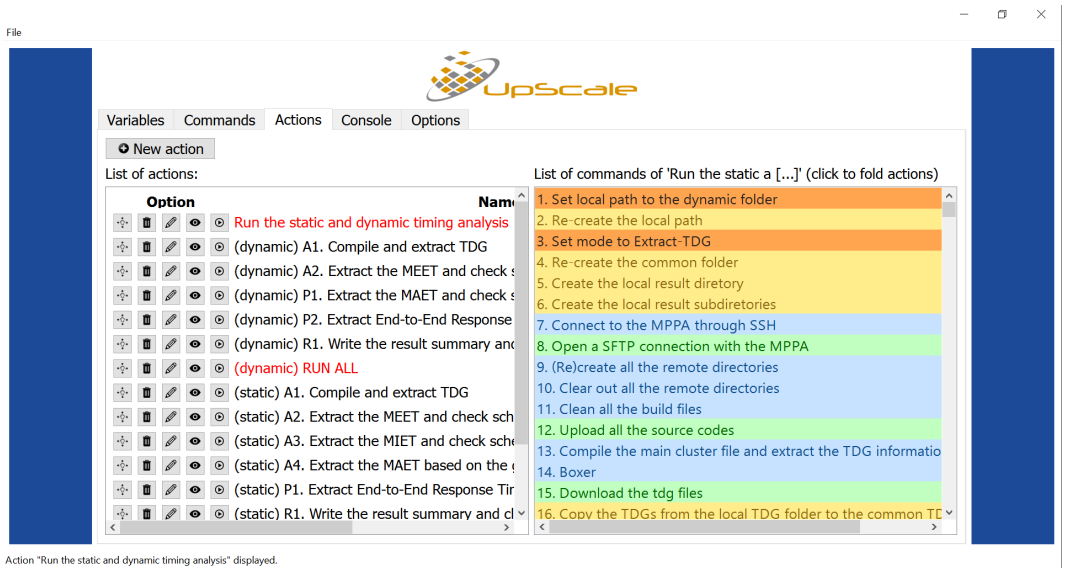


Figure 3 Screenshot of the P-SOCRATES analysis tool-chain that is part of the UpScale SDK.

a Shell script of type "SSH - Remote command[s]", meaning that it will be executed by a remote terminal through SSH. The current version of the tool supports scripts written in Python, R, as well as in any shell script supported by the machine running the script. Files can be sent and received through SSH or SFTP and Shell scripts can be executed remotely through SSH.

The **variables** are defined by the user. They are simply characterised by a name, a type, and a value. Before each command is executed, the tool performs a simple "search and replace" on the code of the command to replace every reference to a variable with its value. Therefore, variables can be used and accessed by every command, irrespective of its programming language, simply by referring to it as "@{variable name}" in the command's code.

The **actions** are the means to connect the commands together. Each action is defined as a set of commands and executing an action simply runs all its commands in the order defined by the user. Note that the tool also provides special control-flow commands that allow to implement loops and simple conditional statements. Those control-flow commands are kept relatively simple as the ambition of the tool is (for now) not to design a new programming language.

6 Results and conclusions

Our timing analysis methodology has been tested on three use-case applications at the end of the project: a pre-processing sampling application for infra-red detectors, an online semantic analysis tool, and a complex event processing engine. Although it is difficult to draw general conclusions from the analysis of these use-cases, the results did provide information which allows to reason on the static and dynamic scheduling and mapping approaches proposed in the project, as well as on the use of the overall analysis flow. It is infeasible in this short paper to summarize the three application use-cases down to a level of details that would allow the reader to verify our conclusions, or to make his own. This is why we simply summarize here the main results that we obtained and the conclusions we made.

One focus of our analysis was on the difference between the MIET, MEET, and MAET of the openMP tasks. That is, we wanted to evaluate the gap between the maximum execution time measured for every openMP task in situations where (a) there is no interference at all on the shared resources (MIET), (b) there is an extreme contention for the shared resources (MEET), and (c) the tasks are executed normally (globally or statically) and their inter-task interference is thus accounted for (MAET).

In [6], we showed that on the Kalray MPPA-256 (Andey) many-core platform the interference generated by concurrent tasks can slow down the execution of a given task by a factor 8, i.e. a task's MEET may be up to 8 times higher than its MIET. In the second half of the P-SOCRATES project, we change the board from a Kalray MPPA Andey to a Kalray MPPA Bostan because the company officially announced that they will no longer support the Andey. Unfortunately, those slow-down factors of 8 could not be reproduced on the Bostan. The reason is that on the Andey, the task interference generated when measuring the MEET of a given task was artificially created by (1) allocating data in the same memory banks as the data of the analyzed task, and then (2) repeatedly accessing those data at runtime. This way we were saturating the controllers of the memory banks used by the task under analysis, thereby slowing down its overall execution. On the Bostan board, we were no longer able to apply this approach because the linker scripts that allowed us (in the Andey) to allocate data to specific banks were not yet implemented at the time on the Bostan.

An interesting observation that we made while analyzing the execution time of the openMP tasks running on the Bostan board is that, by repeatedly calling the built-in "printf()" function from the other cores (that do *not* run the analyzed tasks), we were able to generate a near-starvation scenario. That is, for some tasks, the execution was taking around 49 milli-seconds (with interference) against 117 micro-seconds (without interference) and 124 micro-seconds (when executed normally). This means a slow-down factor or more than 400. Note however, that this result may be due to the internal implementation of the printf function that may temporarily freeze all the cores for some debugging reasons. We have not yet investigated further the reasons for such a slow-down factor.

The results obtained in this analysis allowed to note that static approaches are more time predictable than the dynamic ones. Although for a large number of tasks static mapping and scheduling approaches may experience lower performance at the runtime due to their by-nature conservativeness (thus increasing the actual response time of the application), their prediction on the worst-case response time are tighter and more accurate. The work also leads to conclusions on the need to research on more accurate interference analysis, which leads to less pessimism than worst-case.

References

1 P-SOCRATES Consortium. UpScale-sdk. `http://www.upscale-sdk.com/`. last access 30 Novembre 2016.

2 Kalray Corporation. Kalray. `http://www.kalrayinc.com`. last access 8 April 2015.

3 José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-DAG Model for Real-time Parallel Applications with Conditional Execution. In *30th Annual ACM Symposium on Applied Computing*, pages 1925–1932, Salamanca, Spain, 2015.

4 Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *22Nd International Conference on Real-Time Networks and Systems*, pages 3:3–3:12, Versaille, France, 2014. ACM.

5 A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *27th Euromicro Conference on Real-Time Systems*, pages 211–221, 2015.

6 Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. The Variability of Application Execution Times on a Multi-Core Platform. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OASIcs*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/OASIcs.WCET.2016.6`.

7 P-SOCRATES – Parallel Software Framework for Time-Critical Many-core Systems. `http://www.p-socrates.eu`.

8 L. Pinho, V. Nélis, P. Meumeu Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, and M. Mardiak. P-SOCRATES: a Parallel Software Framework for Time-Critical Many-Core Systems. In *Microprocessors and Microsystems (MICPRO)*, volume 39, Issue 8, pages 1190–1203. Elsevier, 2015.

9 M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. Timing characterization of OpenMP4 tasking model. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166, 2015.