8th International Workshop on Worst-Case Execution Time Analysis

WCET 2008, July 1, 2008, Prague, Czech Republic

Edited by

Raimund Kirner

OASICS

Editor

Raimund Kirner
Real Time Systems Group
Department of Computer Engineering
Vienna University of Technology
Treitlstraße 1–3/182/1
1040 Wien, Austria
`raimund@vmars.tuwien.ac.at`

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

ISBN 978-3-939897-10-1

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

Publication date
November, 2008.

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.2008.i

OASIcs – OpenAccess Series in Informatics

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

ISSN 2190-6807

www.dagstuhl.de/oasics

2008 WCET Abstracts Collection
8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis

Raimund Kirner

Universität Wien, AT
raimund@vmars.tuwien.ac.at

Abstract. The workshop on Worst-Case Execution Time Analysis is a satellite event to the annual Euromicro Conference on Real-Time Systems. It brings together people that are interested in all aspects of timing analysis for real-time systems. In the 2008 edition, 13 papers were presented, organized into four sessions: methods for WCET computation, low-level analysis, system-level analysis and flow-analysis. The workshop was also the opportunity to report from the 2007 WCET tool challenge.

Keywords. Worst-case execution time, real-time systems, timing analysis

2008 WCET Report – Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time Analysis (WCET'08)

Following the successful WCET Tool Challenge in 2006, the second event in this series was organized in 2008, again with support from the ARTIST2 Network of Excellence. The WCET Tool Challenge 2008 (WCC'08) provides benchmark programs and poses a number of "analysis problems" about the dynamic, runtime properties of these programs. The participants are challenged to solve these problems with their program analysis tools. Two kinds of problems are defined: WCET problems, which ask for bounds on the execution time of chosen parts (subprograms) of the benchmarks, under given constraints on input data; and flow-analysis problems, which ask for bounds on the number of times certain parts of the benchmark can be executed, again under some constraints. We describe the organization of WCC'08, the benchmark programs, the participating tools, and the general results, successes, and failures. Most participants found WCC'08 to be a useful test of their tools. Unlike the 2006 Challenge, the WCC'08 participants include several tools for the same target (ARM7, LPC2138), and tools that combine measurements and static analysis, as well as pure static-analysis tools.

Joint work of: Holsti, Niklas; Gustafsson, Jan; Bernat, Guillem; Ballabriga, Clément; Bonenfant, Armelle; Bourgade, Roman; Cassé, Hugues; Cordes, Daniel; Kadlec, Albrecht; Kirner, Raimund; Knoop, Jens; Lokuciejewski, Paul; Merriam, Nicholas; de Michiel, Marianne; Prantl, Adrian; Rieder, Bernhard; Rochange, Christine; Sainrat, Pascal; Schordan, Markus

Keywords: WCET analysis, benchmark

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1663

Towards an Automatic Parametric WCET Analysis

Bygde, Stefan; Lisper, Björn

Static WCET analysis obtains a safe estimation of the WCET of a program. The timing behaviour of a program depends in many cases on input, and an analysis could take advantage of this information to produce a formula in input variables as estimation of the WCET, rather than a constant. A method to do this was suggested in [12]. We have implemented a working prototype of the method to evaluate its feasibility in practice. We show how to reduce complexity of the method and how to simplify parts of it to make it practical for implementation. The prototype implementation indicates that the method presented in [12] successfully can be implemented for a simple imperative language, mostly by using existing libraries.

Keywords: WCET, Flow Analysis, Parametric, Symbolic

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1659

Improving the WCET computation time by IPET using control flow graph partitioning

Ballabriga, Clément; Cassé, Hugues

Implicit Path Enumeration Technique (IPET) is currently largely used to compute Worst Case Execution Time (WCET) by modeling control flow and architecture using integer linear programming (ILP). As precise architecture effects requires a lot of constraints, the super-linear complexity of the ILP solver makes computation times bigger and bigger. In this paper, we propose to split the control flow of the program into smaller parts where a local WCET can be computed faster - as the resulting ILP system is smaller - and to combine these local results to get the overall WCET without loss of precision. The experimentation in our tool OTAWA with lp_solve solver has shown an average computation improvement of 6.5 times.

Keywords: Static analysis, SESE regions, ILP

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1670

Towards Predicated WCET Analysis

Marref, Amine; Bernat, Guillem

In this paper, we propose the use of constraint logic programming as a way of modeling context-sensitive execution-times of program segments. The context-sensitive constraints are collected automatically through static analysis or measurements. We achieve considerable tightness in comparison to traditional calculation methods that exceeded 20% in some cases during evaluation. The use of constraint-logic programming in our calculations proves to be the right choice when compared to the exponential behaviour recorded by the use of integer linear-programming.

Keywords: WCET Analysis, Implicit-Path Enumeration-Technique, Constraint-Logic Programming, Static Analysis

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1667

INFER: Interactive Timing Profiles based on Bayesian Networks

Zolda, Michael

We propose an approach for timing analysis of software-based embedded computer systems that builds on the established probabilistic framework of Bayesian networks. We envision an approach where we take (1) an abstract description of the control flow within a piece of software, and (2) a set of run-time traces, which are combined into a Bayesian network that can be seen as an interactive timing profile. The obtained profile can be used by the embedded systems engineer not only to obtain a probabilistic estimate of the WCET, but also to run interactive timing simulations, or to automatically identify software configurations that are likely to evoke noteworthy timing behavior, like, e.g., high variances of execution times, and which are therefore candidates for further inspection.

Keywords: Bayesian networks, embedded systems, hardware modeling, measurement-based execution time analysis, software modeling, probabilistic modeling, profilin

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1669

Towards a Common WCET Annotation Language: Essential Ingredients

Kirner, Raimund; Kadlec, Albrecht; Prantl, Adrian; Schordan, Markus; Knoop, Jens

Within the last years, ambitions towards the definition of common interfaces and the development of open frameworks have increased the efficiency of research on WCET analysis.

The Annotation Language Challenge for WCET analysis has been proposed in line with these ambitions in order to push the development of common interfaces also to the level of annotation languages, which are crucial for the power of WCET analysis tools. In this paper we present a list of essential ingredients for a common WCET annotation language. The selected ingredients comprise a number of features available in different WCET analysis tools and add several new concepts we consider important. The annotation concepts are described in an abstract format that can be instantiated at different representation levels.

Keywords: WCET, worst-case execution time, hard real-time, embedded systems, abstract interpretation, pipeline analysis, cache analysis, symbolic state traversal BDD

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1657

Computing time as a program variable: a way around infeasible paths

Holsti, Niklas

Conditional branches connect the values of program variables with the execution paths and thus with the execution times, including the worst-case execution time (WCET). Flow analysis aims to discover this connection and represent it as loop bounds and other path constraints. Usually, a specific analysis of the dependencies between branch conditions and assignŋments to variables creates some representation of the feasible paths, for example as IPET execution-count constraints, from which a WCET bound is calculated. This paper explores another approach that uses a more direct connection between variable values and execution time. The execution time is modeled as a program variable. An analysis of the dependencies between variables, including the execution-time variable, gives a WCET bound that excludes many infeasible paths. Examples show that the approach often works, in principle. It remains to be seen if it is scalable to real programs.

Keywords: WCET, flow analysis, infeasible paths, dependency analysis

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1660

Merging Techniques for Faster Derivation of WCET Flow Information using Abstract Execution

Gustafsson, Jan; Ermedahl, Andreas

Static Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems.

A key component in static WCET analysis is to derive flow information, such as loop bounds and infeasible paths. We have previously introduced abstract execution (AE), a method capable of deriving very precise flow information. This paper present different merging techniques that can be used by AE for trading analysis time for flow information precision. It also presents a new technique, ordered merging, which may radically shorten AE analysis times, especially when analyzing large programs with many possible input variable values.

Keywords: Worst-Case Execution Time (WCET) analysis, flow analysis

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1658

On Composable System Timing, Task Timing, and WCET Analysis

Puschner, Peter; Schoeberl, Martin

The complexity of hardware and software architectures used in today's embedded systems make a hierarchical, composable timing analysis impossible. This paper describes the source of this complexity in terms of mechanisms and side effects that determine variations in the timing of single tasks and entire applications. Based on these observations, the paper proposes strategies to reduce the complexity. It shows the positive effects of these strategies on the timing of tasks and on WCET analysis.

Keywords: Real-time systems, timing analysis, WCET analysis, predictable timing, composability

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1662

WCET Analysis for Preemptive Scheduling

Altmeyer, Sebastian; Gebhard, Gernot

Hard real-time systems induce strict constraints on the timing of the task set. Validation of these timing constraints is thus a major challenge during the design of such a system. Whereas the derivation of timing guarantees must already be considered complex if tasks are running to completion, it gets even more complex if tasks are scheduled preemptively – especially due to caches, deployed to improve the average performance. In this paper we propose a new method to compute valid upper bounds on a task's worst case execution time (WCET). Our method approximates an optimal memory layout such that the set of possibly evicted cache-entries during preemption is minimized. This set then delivers information to bound the execution time of tasks under preemption in an adopted WCET analysis.

Keywords: WCET, Preemption

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1664

Applying WCET Analysis at Architectural Level

Gilles, Olivier; Hugues, Jérôme

Real-Time embedded systems must enforce strict timing constraints. In this context, achieving precise Worst Case Execution Time is a prerequisite to apply scheduling analysis and verify system viability. WCET analysis is usually a complex and time-consuming activity. It becomes increasingly complex when one also considers code generation strategies from high-level models. In this paper, we present an experiment made on the coupling of the WCET analysis tool Bound-T and our AADL to code generator OCARINA. We list the different steps to successfully apply WCET analysis directly from model, to limit user intervention.

Keywords: WCET, AADL, Bound-T, Ocarina

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1665

Traces as a Solution to Pessimism and Modeling Costs in WCET Analysis

Whitham, Jack; Audsley, Neil

WCET analysis models for superscalar out-of-order CPUs generally need to be pessimistic in order to account for a wide range of possible dynamic behavior. CPU hardware modifications could be used to constrain operations to known execution paths called traces, permitting exploitation of instruction level parallelism with guaranteed timing. Previous implementations of traces have used microcode to constrain operations, but other possibilities exist. A new implementation strategy (virtual traces) is introduced here. In this paper the benefits and costs of traces are discussed. Advantages of traces include a reduction in pessimism in WCET analysis, with the need to accurately model CPU internals removed. Disadvantages of traces include a reduction of peak throughput of the CPU, a need for deterministic memory and a potential increase in the complexity of WCET models.

Keywords: WCET superscalar cpu virtual traces

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1666

A tool for average and worst-case execution time analysis

Hickey, David Early, Diarmuid; Schellekens, Michel

We have developed a new programming paradigmwhich, for conforming programs, allows the average-case execution time (ACET) to be obtained automatically by a static analysis.

This is achieved by tracking the data structures and their distributions that will exist during all possible executions of a program. This new programming paradigm is called MOQA and the tool which performs the static analysis is called Distritrack. In this paper we give an overview of both MOQA and Distritrack. We then discuss the possibility of extending Distritrack for static worst-case execution time (WCET) analysis ofMOQA programs using the tight tracking of data structures already being performed.

Keywords: Tool, static timing, worst-case, average-case, execution time

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1668

TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis

Prantl, Adrian; Schordan, Markus; Knoop, Jens

TuBound is a conceptually new tool for the worst-case execution time (WCET) analysis of programs. A distinctive feature of TuBound is the seamless integration of a WCET analysis component and of a compiler in a uniform tool. TuBound enables the programmer to provide hints improving the precision of the WCET computation on the high-level program source code, while preserving the advantages of using an optimizing compiler and the accuracy of a WCET analysis performed on the low-level machine code. This way, TuBound ideally serves the needs of both the programmer and the WCET analysis by providing them the interface on the very abstraction level that is most appropriate and convenient to them. In this paper we present the system architecture of TuBound, discuss the internal work-flow of the tool, and report on first measurements using benchmarks from Mälardalen University. TuBound took also part in the WCET Tool Challenge 2008.

Keywords: Worst-case execution time (WCET) analysis, Tool Chain, Flow Constraints, Source-To-Source

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2008/1661

WCET TOOL CHALLENGE 2008: REPORT

Niklas Holsti[1], Jan Gustafsson[2], Guillem Bernat[3] (eds.), Clément Ballabriga[4], Armelle Bonenfant[4], Roman Bourgade[4], Hugues Cassé[4], Daniel Cordes[7], Albrecht Kadlec[5], Raimund Kirner[5], Jens Knoop[6], Paul Lokuciejewski[7], Nicholas Merriam[3], Marianne de Michiel[4], Adrian Prantl[6], Bernhard Rieder[5], Christine Rochange[4], Pascal Sainrat[4], Markus Schordan[6]

Abstract

Following the successful WCET Tool Challenge in 2006, the second event in this series was organized in 2008, again with support from the ARTIST2 Network of Excellence. The WCET Tool Challenge 2008 (WCC'08) provides benchmark programs and poses a number of "analysis problems" about the dynamic, run-time properties of these programs. The participants are challenged to solve these problems with their program-analysis tools. Two kinds of problems are defined: WCET problems, which ask for bounds on the execution time of chosen parts (subprograms) of the benchmarks, under given constraints on input data; and flow-analysis problems, which ask for bounds on the number of times certain parts of the benchmark can be executed, again under some constraints. We describe the organization of WCC'08, the benchmark programs, the participating tools, and the general results, successes, and failures. Most participants found WCC'08 to be a useful test of their tools. Unlike the 2006 Challenge, the WCC'08 participants include several tools for the same target (ARM7, LPC2138), and tools that combine measurements and static analysis, as well as pure static-analysis tools.

1. Introduction

1.1 Worst-Case Execution-Time: Why It is Needed, How to Get It

The chief characteristic of (hard) real-time computing is the requirement to complete the computation within a given time or by a given deadline. The computation or execution time usually depends to some extent on the input data and other variable conditions. It is then important to find

1 Tidorum Ltd, Tiirasaarentie 32, FI 00200 Helsinki, Finland
2 School of Innovation, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden
3 Rapita Systems Ltd, IT Centre, York Science Park, York, YO10 5DG, United Kingdom
4 TRACES group, IRIT - Université Paul Sabatier, 118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9, France
5 Institut für Technische Informatik, Treitlstraße 3/E182.1, Vienna University of Technology, A-1040 Wien, Austria
6 Institut für Computersprachen, Argentinierstraße 8/E185.1, Vienna University of Technology, A-1040 Wien, Austria
7 TU Dortmund University, Department of Computer Science XII (Embedded Systems Group), Otto-Hahn-Strasse 16, 44221 Dortmund, Germany

the *worst-case* execution time (WCET) and verify that it is short enough to meet the deadlines in all cases. For a multi-threaded computation the usual approach is to find the WCETs of each thread and then verify the system timing by some *scheduling analysis*, for example response-time analysis. Finding the WCET of each thread, and of other significant parts of the computation such as interrupt-disabled critical regions, is thus an important step in the verification of a real-time system.

Several methods and tools for WCET analysis have been developed. Some tools are commercially available. The recent survey by Wilhelm *et al.* [42] is a good introduction to these methods and tools. Some tools use pure static analysis of the program; other tools combine static analysis with dynamic measurements of the execution times of program parts. Unlike most applications of program analysis, WCET tools must analyse the *machine* code, not (only) the source code. This means that the analysis depends on the target processor, so a WCET tool typically comes in several versions, one for each supported target processor or even for each target system with a particular set of caches and memory interfaces. Some parts of the machine-code analysis may also depend on the compiler that generates the machine code. For example, the analysis of control-flow in switch-case statements may be sensitive to the compiler's idiomatic use of jumps via tables of addresses.

In general, WCET tools use simplifying approximations and so determine an *upper bound* on the WCET, not the *true* WCET. The pessimism, that is the difference between the true WCET and the upper bound, may be large in some cases. For most real, non-trivial programs a fully automatic WCET analysis is not (yet) possible which means that manual *annotations* or *assertions* are needed to define essential information such as loop iteration bounds. The need for such annotations, and the form in which the annotations are written, depends on both the WCET tool and on the target program to be analysed.

1.2 The WCET Tool Challenge: Aims and History

Several European developers and vendors of WCET tools collaborate in the Timing Analysis sub-cluster of the ARTIST2 Network of Excellence which is financed under Frame Programme 6 of the European Union [3]. Early in this collaboration, Reinhard Wilhelm (Saarland University) proposed a "competition" for WCET tools, modelled on the existing tool competitions, in particular tools for automated deduction and theorem proving (ATP).

However, discussion within the ARTIST2 group found several important differences between ATP tools and WCET tools. Where ATP tools have a standard input format (mathematical logic in textual form), WCET tools use different input formats: different source and machine languages for different instruction sets. Moreover, where ATP tools have a common and unambiguous expected result (a proof or disproof) and the true answer for each benchmark problem is known, WCET tools have a numeric, nearly continuous range of approximate outputs (WCET bounds or WCET estimates) and the true WCET of a benchmark program is often unknown, at least when the target system has caches or other dynamic accelerator mechanisms.

After some discussion, the term "Challenge" was chosen to emphasize that the aim is not to find a "winning" tool, but to challenge the participating tools with common benchmark problems and to enable cross-tool comparisons along several dimensions, including the degree of analysis automation (of control-flow analysis, in particular), the expressiveness and usability of the annotation mechanism, and the precision and safety of the computed WCET bounds. Through the

Challenge, tool developers can demonstrate what their tools can do, and potential users of these tools can compare the features of different tools.

As a result of this discussion, Jan Gustafsson of the Mälardalen Real-Time Centre organized the first WCET Tool Challenge in 2006 [13], using the Mälardalen benchmark collection [26] and the PapaBench benchmark [24], with participation from five tools. The results from WCC'06 were initially reported at the ISoLA 2006 conference [14] and later in complete form [15]. Lili Tan of the University of Duisburg-Essen did an independent evaluation of the tools on these benchmarks, also reported at ISoLA 2006 [36].

The second WCET Tool Challenge was organized in 2008 (WCC'08 [39]) and is the subject of this report. The report combines contributions from the WCC'08 participants and is edited by the WCC'08 steering group, some of whom are also WCC'08 participants. The editors wrote the introductory sections 1 and 2 (the participants contributing the tool descriptions in section 2.4) and the general discussions of problems and results in sections 3 and 4. The participants wrote section 5 to explain how they used their tools to analyse the WCC'08 benchmarks and to discuss their experiences and point out particular problems or successes. The editors wrote the summary and conclusion in section 6.

2. Organization of WCC'08

While WCC'06 was quite a success, it had some significant shortcomings. The main problem was the lack of a common target processor – all tools in WCC'06 assumed different target processors, and so produced incomparable WCET values for the same benchmarks. Another problem was the lack of test suites for the WCC'06 benchmark programs, which made it difficult for measurement-based or hybrid tools to participate – all WCC'06 results used static analysis only. Furthermore, many of these benchmarks were small or synthetic programs with a single execution path, and some were written in non-portable ways that made the analysis unduly difficult on some target processors. Finally, there was no precise definition of the Challenge "problems", that is, which parts of the benchmarks were to be analysed, and under which assumptions, and no common form for presenting the analysis results.

After WCC'06, a working group for the next Challenge was set up and consisted of Jan Gustafsson, Guillem Bernat, and Niklas Holsti. This became the steering group for WCC'08, with the aim of correcting the shortcomings of WCC'06: choosing a common target processor; choosing portable benchmark programs; creating test suites to let measurement-based tools participate; defining the analysis problems precisely enough to ensure that all participants make the same analyses; and defining a common result format for easy comparison. We were fortunate to get some financial support from ARTIST2 for this work.

The WCC'08 schedule was intended to produce results for presentation at the annual WCET Workshop, which in 2008 took place on July 1. However, the planning and structuring of the Challenge took some time to converge. The final version of the "debie1" benchmark, which became the main benchmark, was not available until April 9. Moreover, some participants had to port their tools to the ARM7 target which also caused delay. In the end, only one participant (OTAWA) produced results before the WCET Workshop; the rest entered their results later. For WCC'08 the participants (tool developers) did all their own analyses. There was no “independent” analysis as done by Lili Tan for WCC'06 [36].

The main tool for organizing WCC'08 was the Wiki site hosted at Mälardalen [40]. We thank Hüseyn Aysan of MRTC for setting up the Wiki framework. Most of the initial Wiki content was written by Niklas Holsti with ARTIST2 funding. The Wiki defines the benchmark programs and the analysis problems – in other words, the questions that the participants should answer with their tools. For each benchmark program there is a page with result tables in which WCC'08 participants enter their results. WCC'08 participants have full editing access to the Wiki; non-participants have read-only access. In addition to the benchmarks, analysis problems, and result tables, the Wiki contains sundry information about the target processors and cross-compilers and a collection of questions and answers that arose during WCC'08.

2.1 The WCC'08 Benchmarks

For WCC'08 we wanted benchmark programs that are relatively large, preferably real programs, or based on real programs, rather than synthetic ones, and are provided with test suites for measurement-based WCET analysis. We also wanted new benchmarks to make a change from WCC'06. In the end, because of labour constraints, none of the WCC'06 benchmarks were included in WCC'08. We regret in particular the absence of PapaBench [24], caused by the lack of a test suite, and hope that it will be included in a future Challenge. WCC'08 defined five benchmarks: the “debie1” benchmark, courtesy of Space Systems Finland Ltd (SSF), and four benchmarks contributed by Rathijit Sen and Reinhard Wilhelm of Saarland University. The rest of this section briefly describes these benchmarks.

The “debie1” benchmark

The “debie1” benchmark is based on the on-board software of the DEBIE-1 satellite instrument for measuring impacts of small space debris and micro-meteoroids [11]. The software is written in C, originally for the 8051 processor architecture, specifically an 80C32 processor that is the core of the the Data Processing Unit (DPU) in DEBIE-1. The software consists of six tasks (threads). The main function is interrupt-driven: when an impact is recorded by a sensor unit, the interrupt handler starts a chain of actions that read the electrical and mechanical sensors, classify the impact according to certain quantitative criteria, and store the data in the SRAM memory. These actions have hard real-time deadlines that come from the electrical characteristics (hold time) of the sensors. Some of the actions are done in the interrupt handler, some in an ordinary task that is activated by a message from the interrupt handler. Two other interrupts drive communication tasks: telecommand reception and telemetry transmission. A periodic housekeeping task monitors the system by measuring voltages and temperatures and checking them against normal limits, and by other checks. The DEBIE-1 software and its WCET analysis with Bound-T were described at the DASIA'2000 conference [17]. The WCC'08 Wiki also has a more detailed description [40].

The real DEBIE-1 flight software was converted into the “debie1” benchmark by removing the proprietary real-time kernel and the low-level peripheral interface code and substituting a test harness that simulates some of those functions. Moreover, a suite of tests was created in the form of a test driver function. The benchmark program is single-threaded, not concurrent; the test driver simulates concurrency by invoking thread main functions in a specific order. The DEBIE-1 application functions, the test harness, and the test driver are linked into the same executable. This work was done at Tidorum Ltd by Niklas Holsti with ARTIST2 funding.

SSF provides the DEBIE-1 software for use as a WCET benchmark under specific Terms of Use that do not allow fully open distribution. Therefore, the "debie1" benchmark is not directly downloadable from the WCC'08 Wiki. Copies of the software can be requested from Tidorum[8]. SSF has authorized Tidorum to distribute the software for such purposes.

The benchmarks "rathijit_1" through "rathijit_4"

The goal of this group of benchmark programs, constructed and contributed by Rathijit Sen and Reinhard Wilhelm of Saarland University, is to stress-test both instruction-cache and data-cache analysers. The programs use a large number of C macros within loops. When these macros are instantiated by the C preprocessor the code size becomes quite large. The intent is to have a large instruction-cache footprint. The large code size means that these benchmarks also test the scalability of all analyses. There is no particular rationale or meaning behind choosing any particular constant, construct, condition, or ordering. They have been chosen at random. The programs are single-threaded.

The "rathijit_1" benchmark tests data reuse and control-flow analysis by initializing and walking through various parts of a 2D array. The "rathijit_2" benchmark aims to test control flow analysis, data-flow analysis, data reuse, and alias analysis. The basic unit has a 2-nested loop within a 3-nested conditional check and accesses some portion of 4 arrays through pointers which are set depending on the arguments passed to the basic unit. Globally there are 4 2D arrays, and the basic unit is instantiated a number of times. The goal of "rathijit_3" is to test code and data reuse in functions across different invocations. The program contains a 2-dimensional grid of functions, *func_i_j*, with *i* and *j* in 0 .. 10, for a total of 121 functions. The number of times each function is called depends on its position in the grid. Thus, *func_i_j* is called a number of times depending on *i* and *j* and should be able to reuse code and data fetched earlier, provided they have not been evicted from the cache. It is also possible that the data could have been fetched by some other function. The "rathijit_4" benchmark again tests control-flow analysis and data reuse by 4-nested switch-case statements, conditional branches and function calls from within the case statements.

The "rathijit" benchmarks are published under a liberal open-source licence and can be downloaded directly from the WCC'08 Wiki site [40].

2.2 The WCC'08 Analysis Problems

For each WCC'08 benchmark a number of *analysis problem*s or questions are defined, for the participants to analyse and answer. There are two kinds of problems: WCET-analysis problems and flow-analysis problems. Flow-analysis problems can be answered by tools that focus on flow-analysis (for example SWEET [35], unfortunately not a WCC'08 participant) but that do not have the "low-level" analysis for computing WCET bounds (for the ARM7 processor, or for any processor). Flow-analysis problems can also show differences in the flow-analyses of different WCET tools, and this may explain differences in the WCET bounds computed by the tools.

8 niklas.holsti@tidorum.fi

A typical WCET-analysis problem asks for bounds on the WCET of a specific subprogram within the benchmark program (including the execution of other subprograms called from this subprogram). For example, problem 4a-T1 for the “debie1” benchmark asks for the WCET of the *HandleTelecommand* function when the variable input data satisfy some specific constraints.

A typical flow-analysis problem asks for bounds on the number of times the benchmark program executes a certain statement, or a certain set of statements, within one execution of a root subprogram. For example, problem 4a-F1 for the “debie1” benchmark asks how many calls of the macro *SET_DATA_BYTE* can be executed within one execution of the function *HandleTelecommand*, under the same input-data constraints as in the WCET-analysis problem 4a-T1. By further requiring the analysis to assume that the execution time of *SET_DATA_BYTE* is arbitrarily large we make it possible for pure WCET-analysis tools to answer this flow-analysis question, since this assumption forces the worst-case path to include the maximum number of *SET_DATA_BYTE* calls; all alternative paths have a smaller execution time.

2.3 The WCC'08 Suggested Common Target Processor

After polling the potential participants, we decided to suggest the ARM7 processor as the common target for WCC'08. However, other targets were also allowed; the TuBound group used the C167. Since different ARM7 implementations may have different timing for memory accesses we picked a particular ARM7 chip, the LPC2138 from NXP Semiconductor [29]. The IF-DEV-LPC kit from iSYSTEM [18] was recommended for running LPC2138 benchmarks and was used by Tidorum and Rapita Systems. However, iSYSTEM no longer supply this kit in single quantities. The MTime group used another board, from OLIMEX [27]. The execution times should be the same on all LPC2138 boards because all timing interactions are on-chip and involve no off-chip components.

The ARM7 is basically a simple, deterministic processor that does not challenge the analysis of caches and complex pipelines that are important features of some WCET tools [42]. We had to choose such a simple common target because a more complex one would have required a large effort from most participants. Even so, some potential participants withdrew from WCC'08 because they did not have time to port their tools to the ARM7. More complex targets are under consideration for future Challenges and were certainly not excluded from the invitation to WCC'08.

The ARM7 architecture

The ARM7 [2] is a 32-bit pipelined RISC architecture with a single (von Neumann) address space. All basic ARM7 instructions are 32 bits long. Some ARM7 devices support the alternative THUMB instruction set, with 16-bit instructions, but this was not used in WCC'08. The ARM7 processor has 16 general registers of 32 bits. Register 15 is the Program Counter. Thus, when this register is used as a source operand it has a static value, and if it is a destination operand the instruction acts as a branch. Register 14 is designated as the “link register” to hold the return address when a subprogram call occurs. There are no specific call/return instructions; any instruction sequence that has the desired effect can be used. This makes it harder for static analysis to detect call points and return points in ARM7 machine code. The timing of ARM7 instructions is basically deterministic. Each instruction is documented as taking a certain number of “incremental” execution cycles of three kinds: “sequential” and “non-sequential” memory-access cycles and “internal” processor cycles. The actual duration of a memory-access cycle can depend on the memory subsystem. The term “incremental” refers to the pipelining of instructions, but the pipeline is a simple linear one, and the total execution-time of an instruction sequence is generally the sum of the incremental times of the instructions

The LPC2138 chip and the MAM

The NXP LPC2138 implements the ARM7 architecture as a microcontroller with 512 KiB of on-chip flash memory starting at address zero and usually storing code, and 32 KiB of static on-chip random-access memory (SRAM) starting at adress 0x4000 0000 and usually storing variable data. There is no off-chip memory interface, only peripheral I/O (including, however, I2C, SPI, and SSP serial interfaces that can drive memory units).

The on-chip SRAM has a single-cycle (no-wait) access time at any clock frequency. The on-chip flash allows single-cycle access only up to 20 MHz clock frequency. At higher clock frequencies, up to the LPC2138 maximum of 60 MHz, the flash needs wait cycles. This can delay instruction fetching and other flash-data access, but the LPC2138 contains a device called the *Memory Acceleration Module* (MAM) that reduces this delay by a combination of caching and prefetching as follows.

The flash-memory interface width is 128 bits, or four 32-bit words. The MAM contains three 128-bit buffers that store, or cache, flash contents: the *Prefetch* buffer, the *Branch Trail* buffer, and the *Data* buffer. When the MAM is enabled, the Prefetch buffer holds the 128-bit block that contains the current instruction. The MAM concurrently prefetches the next 128-bit flash block into a fourth internal buffer called the “latch”. If execution continues sequentially from the last instruction in the Prefetch buffer, the next instructions are often already present in the MAM latch and are then moved to the Prefetch buffer so that execution continues without delay. The MAM again starts to prefetch the next 128-bit block from the flash to the latch. Instructions that read data from the flash make the MAM abort the prefetch, read the requested data from the flash while the processor waits, and restart the prefetch. This can force the processor to wait also for the next 128-bit instruction block. The MAM Data buffer caches the most recently read 128-bit block of flash data. This benefits sequential data access but is not useful for random access.

When a branch occurs, the processor must generally wait for the MAM to read the 128-bit block that contains the target instruction. The Branch Trail buffer in the MAM holds the last 128-bit block that has been the target of a branch. Thus, when an innermost loop has no internal branches, the loop-repeating branch usually “hits” in the Branch Trail buffer and causes no fetch delays.

The effect of the MAM on WCET analysis is similar to that of a cache. The state of the MAM depends on the dynamic history of the execution, and the state affects the time for instruction fetch and data access from the flash memory. The flash memory is divided into 128-bit blocks, similar to cache lines, and execution-timing can depend on which block an instruction or a datum lies in, and whether it lies in the flash or in the SRAM. For branches the timing can also depend on the offset of the source and target instructions within their blocks, through the block-prefetch function.

MAM configurations m2t1 and m2t3

In the available version of the LPC2138 the MAM has two bugs [25] that limit the useful MAM configurations to Mode 2 (MAM fully enabled). However, when the number of cycles for a flash access is set to 1 the MAM has no effect on timing, as if the MAM were disabled. We defined two MAM configurations for WCC'08: Mode 2 with 1 access cycle, and Mode 2 with 3 access cycles. These configurations are abbreviated *m2t1* and *m2t3*. In *m2t1* the MAM has no effect and timing is static. In *m2t3* the MAM has a dynamic effect on timing as discussed above.

Other factors that affect execution timing on the LPC2138

The on-chip peripherals in the LPC2138 connect to a VLSI Peripheral Bus (VPB) which connects to the Advanced High-performance Bus (AHB) through an AHB-VPB bridge. This bus hierarchy causes some delay when the ARM7 core accesses a peripheral register through the AHB. If the VPB is configured to run at a lower clock frequency than the ARM7 core this delay is variable because it depends on the phase of the VPB clock when the access occurs.

The programming tools

The IF-DEV-LPC kit from iSYSTEM comes with an integrated development environment called WinIDEA and a GNU cross-compiler and linker. The distributed benchmark binaries for WCC'08 were created with Build 118 of these tools using *gcc*-4.2.2 [19]. The IF-DEV-LPC kit has an USB connection to the controlling PC and internally uses JTAG to access the LPC2138. WinIDEA supports debugging with breakpoints, memory inspections, and so on.

The MTime group used another kit, the OLIMEX LPC-H2138 board [27] with *openocd* and *gcc*-4.2.1 as the development environment. This board has a serial I/O port which the MTime group found useful for software instrumentation techniques.

2.4 Tools Participating in WCC'08

The tools that participated in WCC'08 are (in alphabetical order) Bound-T, MTime, OTAWA, RapiTime, TuBound, and the WCET-aware C compiler wcc. Of these, only Bound-T and MTime also participated in the first Challenge in 2006. OTAWA, TuBound, and wcc are new tools and were not ready for use in 2006. RapiTime is a hybrid tool that needs a test suite for each benchmark, and such suites were not available for the WCC'06 benchmarks. Short descriptions of the WCC'08 participating tools follow, in alphabetical order by tool name.

Bound-T from Tidorum Ltd

Bound-T [37] is a typical static WCET analysis tool for simple processors. It loads an executable file, decodes instructions to build control-flow graphs and the call-graph, uses data-flow analysis to find (some) loop bounds, and IPET (per subprogram) to find WCET bounds. The data-flow analysis models the computation with Presburger arithmetic as explained in [16]. An assertion (annotation) language is provided. There is no cache analysis or other support for complex, dynamic processors. Bound-T is at present a closed-source tool that Tidorum licenses to users.

Bound-T is implemented in Ada. It uses no special formalisms to describe target processors or analysis algorithms, and no generative methods, but does use generic Ada modules. For example, a generic least-fixpoint solver is instantiated for various data domains and transfer functions to create a constant propagator, a def-use analyser, and other data-flow analyses.

An early version of Bound-T was used to analyse the original DEBIE-1 program [17] from which the WCC'08 benchmark "debie1" is derived. Moreover, the principal author of Bound-T (Holsti) took part in the creation of the DEBIE-1 program and also converted that program into the "debie1" benchmark for WCC'08 and defined the analysis problems for the benchmark. Thus Bound-T and Tidorum may have had some head start on the analysis of this benchmark for WCC'08.

MTime from Vienna University of Technology

MTime is a measurement based execution time analysis tool. It uses static analysis to split the program in smaller program segments (PS) with a manageable number of execution paths and uses model checking to generate test data [41]. The new version of the tool, which is currently under development, is also able to determine loop bounds by model checking [33]. Unfortunately, due to time constraints we were not able to prepare the new version for WCC'08. Thus we used the stable version of MTime within WCC'08, which is not able to analyze programs with loops.

The test data generated by MTime are used to enforce the execution of all feasible paths within each PS to perform the execution-time measurement. In the following calculation step the measured execution times are combined and an estimate for the WCET is calculated. Mutually exclusive paths within each PS are excluded but pessimism can be introduced by mutually exclusive paths spanning over more than a single PS. The obtained WCET estimate can be guaranteed on simple hardware without dynamic run-time optimization or run-time resource allocation, when all instructions have constant execution time and the CFGs of the source code and object code are the same.

The aim of the tool is to provide a convenient and platform-independent way to perform execution-time measurements. Since the tool operates on source code and uses a modular design, which makes it possible to use different instrumentation and measurement techniques by loading different modules, it is virtually platform independent.

OTAWA from the TRACES group at IRIT

OTAWA [8, 28] is a framework dedicated to the development of WCET analyzers. It includes a range of facilities like:

- loaders (to load the binary code to be analyzed – several ISAs are supported –, flow facts, a description of the hardware, *etc*.),
- code analyzers (*eg*. a CFG builder),
- code processors that perform specific analyses (*eg*. pipeline analysis, cache analysis, branch prediction analysis, etc.) and attach some properties to code items (*eg*. a time value can be attached to each basic block, a cache access category can be attached to an instruction, *etc*.)
- an IPET module that builds the IPET formulation from the CFG, the flow facts and the results of the invoked code processors.

OTAWA is an open software and has been designed to make the integration of new code processors easy. It is available under the LGPL licence.

RapiTime from Rapita Systems Ltd

RapiTime [32] is a measurement-based tool, *i.e.*, it derives timing information of how long a particular section of code (generally a basic block) takes to run from measurements. Timing information is captured on the running system by either a software instrumentation library, a lightweight software instrumentation with external hardware support, purely nonintrusive tracing mechanisms (like Nexus and ETM) or even traces from CPU simulators. The user has to provide test data from which measurements will be taken. Measurement results are combined according to the structure of the program to determine an estimate for the longest path through the program. The program structure is a tree that is derived from either the source code or from the direct analysis of executables. The user can add annotations in the code to guide how the instrumentation and analysis process will be performed, to bound the number of iterations of loops, *etc*.

RapiTime aims at medium to large real-time embedded systems on advanced processors. The tool does not rely on a model of the processor. Thus, in principle, it can model any processing unit (even with out-of-order execution, multiple execution units, various hierarchies of caches, *etc.*). Adapting the tool for new architectures requires porting the object code reader (if needed) and determining a tracing mechanism for that system. RapiTime is the commercial-quality version of the pWCET tool developed at the Real-Time Systems Research Group at the University of York [5].

TuBound from Vienna University of Technology

TuBound is a research prototype of a WCET-analysis and program-development tool-chain [30]. A distinctive feature of TuBound is that it allows to annotate programs with flow information at the source code level. This flow information is then transformed conjointly to code transformations within the development tool chain. TuBound currently includes a C++ source-to-source transformer, a static analysis component, a WCET-aware C compiler, and a static WCET analysis tool. The WCET analysis tool currently integrated into the TuBound tool-chain is *calc_wcet_167*, a static WCET analysis tool that supports the Infineon C167 as target processor.

WCET-aware C compiler wcc from Dortmund University of Technology

In contrast to other tools participating in this challenge, the *wcc* [12] is not a pure WCET analyzer but a complex compiler framework allowing an automatic WCET minimization. The WCET-aware compiler has been used for the development of different compiler optimizations driven by WCET information. Examples for WCET-aware optimizations are Procedure Cloning [23] or a cache-based Procedure Positioning [22].

The compiler consists of a high-level intermediate representation (IR), called *ICD-C IR*, and a low-level IR, called *LLIR*. This separation of the code representation offers potential for a wide range of different analyses and optimizations. The currently supported target hardware is the Infineon TriCore 1.3, a 32-bit microcontroller-DSP architecture optimized for high performance real-time embedded systems. The processor supports different memory hierarchies including caches, scratchpad memories and flashes.

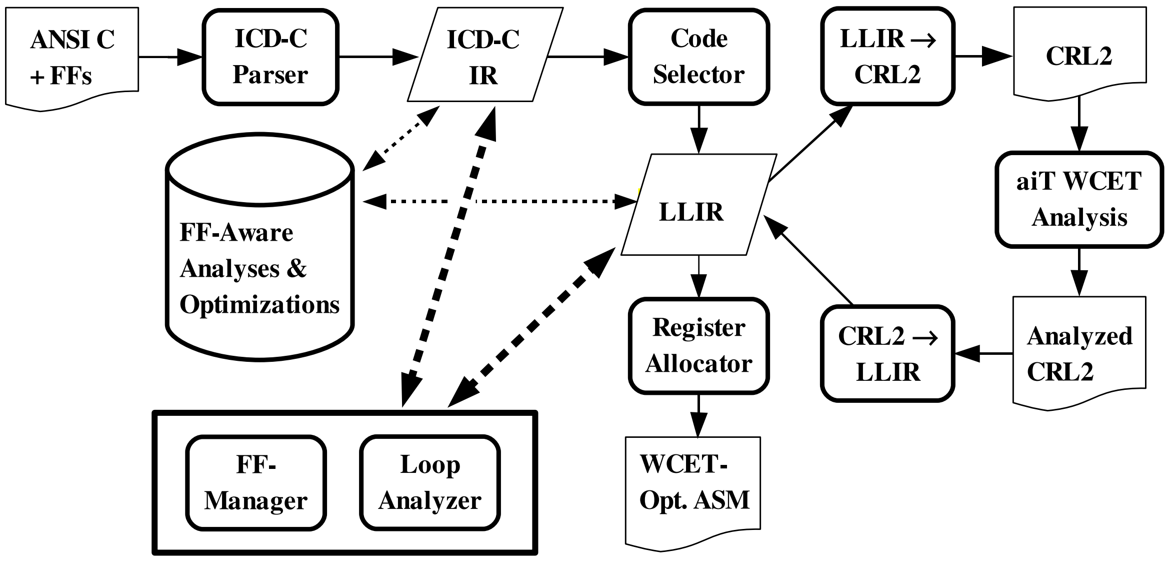


Figure 1: Workflow of wcc

The workflow is depicted in Figure 1. After parsing the C code, possibly annotated with flow facts (FF), into the ICD-C IR, standard compiler analyses, like data- and control-flow analyses, might be performed. In addition, the user can select from a large set of more than 30 average-case execution time (ACET) and WCET optimizations. The generation of the low-level IR is performed by the *code selector*. At the LLIR level, the user can choose again between standard analyses and numerous ACET and WCET optimizations. The code can be finally dumped into an assembly file and passed to the assembler.

Besides the typical components of an optimizing compiler, the *wcc* is extended by various WCET concepts. The fundamental extension distinguishing our compiler from other compilers is the binding of the compiler back-end with the static WCET analyzer *aiT* [1] developed by the company AbsInt. In a first step, the LLIR code is converted into CRL2, a machine-code intermediate representation, and used as input for the automatic invocation of aiT. After the WCET analysis, WCET and execution count information is imported back into the compiler and can be exploited for further applications, e.g. low-level WCET-driven optimizations.

The static loop analyzer [9] is another crucial module to turn a static WCET analysis framework into a fully automatic system. *wcc*'s loop analysis operates on the high-level IR and is based on Abstract Interpretation and polytope models. In addition, a technique called Program Slicing is deployed to accelerate the static analysis. The loop analyzer computes loop bounds and dumps them either in a human-readable format or passes them to the *Flow Fact (FF) Manager.*

The FF Manager is responsible for keeping flow facts, which are either read from the source code or generated by the loop analyzer, consistent. During optimizations, like Loop Unrolling, the original flow facts might become invalid. The manager keeps track of applied optimizations and automatically adjusts flow facts if required. In addition, the manager is responsible for the translation of ICD-C flow facts into the LLIR and further into CRL2. Thus, at any level of the code representation and after the application of optimizations, flow facts are correct.

To sum up, the *wcc* is a framework offering on the one hand standard compiler functionalities to translate an optimized C source code into machine code. On the other hand, the described extensions provide an automatic static WCET estimation of high- and low-level code for a state-of-the-art processor, compiler optimizations tailored towards an automatic WCET minimization and a static computation of flow facts. The latter is deployed to generate flow analysis results reported in the WCC'08.

3. Problems and Solutions

3.1 Life as Usual in the Embedded World

As usual for embedded programming, the participants that actually tried to run the benchmarks on a real processor had various problems with the development tools. For example, some versions and configurations of WinIDEA led to executables that did not initialize the global variables correctly; the bugs in the LPC2138 MAM [25] made some programs run incorrectly under some MAM configurations; and some participants experienced flash “lock-ups” that prevented reprogramming of the flash memory. Solutions to these problems were found by avoiding MAM modes 0 and 1 [25]; upgrading the development tools [19]; and using correct settings in obscure tool menus.

3.2 Problems in the Analysis

As these benchmarks are new ones, not analysed before by any WCC'08 participant, some problems were to be expected. Although the DEBIE program had been analysed for WCET earlier [17], that was for a different target – the Intel 8051, or 80C32 to be exact – not for the ARM7. In fact, for some participants this was their first extensive analysis of ARM7 code: the MTime and OTAWA groups ported their tools to the ARM7 target for WCC'08, and the ARM7 version of Bound-T, initially a mere prototype, was significantly extended for WCC'08.

Even so, most problems reported by the participants came not from the WCET tools, but from the wording of the definitions of the WCC'08 analysis problems or questions, some of which assume rather complex execution scenarios and constraints. The complexity was intentional, to stress the capability of the annotation languages. Indeed no tool was able to implement all constraints as flow-fact annotations. The obscurities and omissions in the definitions were not intentional.

4. Results

4.1 Tools and Targets

Table 1 below shows which target processors each participant has addressed for WCC'08 (most tools support other target processors, too). A notable fact is that four of six tools do flow analysis on the source-code level. This means that their flow-analysis results could in principle be compared in source-code terms. For example, on the source-code level we can talk about iteration bounds for specific loops, which is not possible on the machine-code level because of code optimizations. Future Challenges should perhaps pose flow question also on the source-code level. Another fact shown in Table 1 is that the suggested common target processor, the ARM7 LPC2138, is indeed supported by many participants, at least in the simple *m2t1* mode. Note also that although *wcc* was used in WCC'08 only for source-code flow-analysis, it can produce WCETs for its target processors.

Table 1: Tools and targets in WCC'08

	Source-code flow analysis	***LPC2138 m2t1***	***LPC2138 m2t3***	***C167***
Bound-T		+		
MTime	+	+	+	
OTAWA	+	+	+	
RapiTime		+	+	
TuBound	+			+
wcc	+			

4.2 Results

The full set of results is too large to be presented here; please refer to the Wiki [40]. Table 2 below shows the number of analysis problems for each WCC'08 benchmark, the number of flow-analysis and WCET-analysis questions to be answered, and the number of questions answered by each participating tool (by 10 September 2008). If a tool answers the same question for several target processors, or for several MAM modes on the LPC2138, it still counts as only one answer. The absence of results, so far, for MTime and RapiTime is explained in section 5. Note that at present it is not certain that all participants have interpreted the analysis problems in exactly the same way (same input and execution constraints) which means that the current results (on the Wiki) from different tools may not be comparable even for the same target. We are working on this.

Table 2: Number of posed and answered analysis problems in WCC'08

Posed problems and questions										
Benchmark	***debie1***		***rathijit_1***		***rathijit_2***		***rathijit_3***		***rathijit_4***	
Number of problems	18		2		6		1		2	
Type of question	***Flow***	***WCET***	***Flow***	***WCET***	***Flow***	***WCET***	***Flow***	***WCET***	***Flow***	***WCET***
Number of questions	15	22	1	2	6	6	1	1	2	2
Answered questions (blank = none)										
Bound-T	13	18					1	1		
MTime										
OTAWA	7	16			2					
RapiTime										
TuBound	11	18		1		1		1		1
wcc	15		1		6		1		2	

WCC'08 actually has two kinds of analysis questions for WCET. The more common kind asks for the WCET of a given subprogram. The less common kind asks for the WCET of the interrupt-disabled regions within a given subprogram, which is an an important input to schedulability analysis. However, no participant answered questions of the second kind, although such problems have been studied before [7].

5. Tools and Experiences

This section collects the participants' reports on their goals, problems, solutions, and other comments on WCC'08. It is divided into a subsection for each participating tool, written by the developers of that tool, edited only for uniform formatting.

5.1 Bound-T (written by N. Holsti)

Adapting Bound-T to the ARM7

The basic ARM7 architecture is well suited to Bound-T. The only troublesome aspect is the lack of dedicated call and return instructions which makes it harder to split the target program into subprograms based on the machine code alone. For the ARM7 version of Bound-T, I took the timing of each ARM7 instruction from the “incremental” cycle numbers in the ARM7 manual [2]. The execution time of an instruction sequence is taken to be the sum of these numbers. In other words, I assumed that the incremental numbers include all possible pipeline stalls. I verified this timing model by measuring the execution time of suitable parts of the “debie1” benchmark on an IF-DEV-LPC kit. For these measurements, the software was instrumented with instructions to set the value of an LPC2138 output port (P0), the port was connected to a logic analyzer that recorded the value of the port on each clock, and the number of clock cycles between instrumentation points was computed from this record.

The value-analysis part of Bound-T at first had problems with the ARM7 code from *gcc*, because *gcc* often uses different registers as temporary pointers to stack variables. I extended the constant-propagation part of the value-analysis to handle semi-symbolic values of the form $P + c$, where P is a symbol representing the initial value of the stack pointer (the base of the stack frame for the subprogram under analysis) and c is a constant. References to such memory addresses are then resolved into references to statically identified parameters or local variables, even though the actual value of P is unknown. This extension will be useful for other targets, too, in particular when the compiler uses frame pointers for some, but not all subprograms.

Problems with the LPC2138

Bound-T has no general cache analysis, but I planned to model the LPC2138 MAM using a specific abstract interpretation. However, while the MAM at first seems rather simple, from the description in [29], with more study one quickly finds undocumented areas. For example, the detailed timing of aborting and restarting an instruction prefetch is not clear. To investigate these questions I made several test programs that systematically test many cases, for example branches with all 16 combinations of 128-bit-block offsets of the branch instruction and the target instruction. The test programs were densely instrumented with port outputs and measured with the logic analyzer. I found some interesting results, such as loops in which the execution time of the loop body alternates between two values – faster on every other iteration, slower on the rest – even when every iteration executes the same instructions. I found no simple conceptual model that could explain the MAM timing. In desperation I tried to make a cycle-accurate simulation of the MAM, in the form of clocked registers and logic elements. This model accurately predicted most, but not quite all, of the observations. Running out of time and energy, I abandoned the MAM modelling, so for WCC'08 Bound-T/ARM7 can analyse only the *m2t1* configuration, where the MAM has no effect on timing.

Another, smaller problem with the LPC2138 is the location of the SRAM at the relatively large address of 0x4000 0000. The value-analysis in Bound-T depends on an external tool (the Omega Calculator [31]) that uses 32-bit integers and conservatively checks against overflow. However, this tool sometimes aborted the value-analysis of pointers into the SRAM because intermediate results threatened to overflow. I worked around this problem with annotations or by linking the benchmark programs with a memory map that uses a smaller SRAM address.

Problems and successes with the benchmarks

The Bound-T analysis of the "debie1" benchmark went in general as well as could be expected: many loop-bounds were found automatically, but many were not. Some of the WCC'08 execution constraints could be expressed in the Bound-T annotation language, but some could not, or required ugly work-arounds. For some "disjunctive" constraints I had to make several analyses with different annotations and combine the results manually. This is described in detail in the Bound-T notes on the WCC'08 Wiki site, including all annotations that I used. Of course, I had an unfair advantage over the other WCC'08 participants for this benchmark, because of my earlier association with the development of the DEBIE-1 software [17] and my work on the definition of the "debie1" benchmark.

Bound-T was much less successful with the "rathijit" benchmarks. The subprograms in "rathijit_1" and "rathijit_2" are too large for the value-analysis in Bound-T, which did not terminate in a reasonable time, or ran out of memory. Annotations can sometimes avoid the need for this value-analysis, but are no help in this case because Bound-T needs value-analysis to analyse the dynamically branching code from the numerous switch-case statements. Also, the large number of loops in these benchmarks would make manual annotation cumbersome. In "rathjiit_3" the compiled forms of the loops terminate by comparing the values of pointers to the LPC2138 SRAM area, and these values are too large for the value analysis in Bound-T. Relinking the program with a smaller SRAM address enabled successful automatic analysis of "rathijit_3". In "rathijit_4", the subprogram *func1* is compiled to an irreducible control-flow graph. The current version of Bound-T cannot analyze loops in irreducible graphs.

Comments on the WCET Tool Challenge

Considering WCC'08 as a participant, not as one of the organizers, I found it quite useful for driving the ARM7 version of Bound-T from a prototype towards a practical tool. The comparison of the Bound-T results to similar tools (in particular OTAWA) was a good check. The "rathijit" benchmarks will be good measures of future progress in the scalability of Bound-T. As an organizer, I enjoyed the contacts and discussions with all the participants.

5.2 MTime (written by B. Rieder and R. Kirner)

This section presents the results of the MTime tool for the WCET Tool Challenge 2008. Although it has not been possible to measure WCET results for the given benchmarks there are some results.

Target Hardware

First, a new instrumentation and measurement module for the old version of the MTime framework was created to support the ARM7TDMI platform. To perform execution-time measurements the OLIMEX ARM-USB-OCD Programmer and the LPC-H2138 development board were used [27]. The used target hardware provides an RS232 port on the development board and the programmer provides an ARM JTAG interface and an USB-to-RS232 converter on a single USB connector, which makes it the ideal choice for laptops or PCs without a serial port. An additional advantage is that the hardware is fully Linux compatible and fully supported by the *openocd* on-chip debugger while iSYSTEM, the manufacturer of the ITAG-U-ARM programmer and NXP LPC2138-M Mini-Target Board [18], neither supplies Linux software nor supplies information about the interface protocol of the ITAG-U-ARM JTAG programmer.

Measurements

The execution-time measurements are performed using the internal timer T0 of the LPC2138 which is set to run with the full CPU frequency. The measurement starts at the beginning of a PS with four assembly instructions which write “1” to the TCR0 register, which starts the timer, and ends at the end of the PS with another four instructions, writing a “0” to the TCR0 register for the timer to stop. The register width is 32 bits and there is an overflow register which is also 32 bits wide, resulting in a maximum counter value of 2^{64}, which should be sufficient for all applications. The current implementation uses a small boot loader, which is located in the flash, to download the application over the RS232 port. The test data and the measurement results are also transferred using the RS232 connection. The measured program resides in the SRAM. For this reason the MAM problem does not arise, but results are likely to be different from flash-based solutions.

WCET Tool Challenge Benchmarks

The stable version of the MTime framework does not support loops, therefore no benchmarks from the challenge could be performed.

Comments on the WCET Tool Challenge

The challenge is a very good opportunity to compare current execution time analysis frameworks. To increase the comparability of the individual tools, we propose to add more synthetic benchmarks, with small isolated problems. Benchmarks should be ordered by increasing complexity starting from single-path code and ending in applications with nested loops with data-dependent control flow. Adding complexity gradually would also increase the comparability of the results and point out individual weaknesses or strengths of individual analysis tools, providing valuable information for the both tool developers and users. Additionally, it would be interesting to measure not only the quality of the WCET bound but also the effort required for the preparation of the analysis (how much time to add annotations, *etc.*) and the time required for the analysis.

5.3 OTAWA (written by the TRACES group, see footnote 4 on title page)

General problems

The main problem we encountered is related to the specification of the "debie1" problems. First, we found it difficult to understand the meaning of some of the constraints and then we could not see how they should be translated into flow facts. We believe that defining a language to express the constraints would help, but this means that the constraints should probably be specified at a lower level (*eg*. line *x* in the *foo.c* source file – or instruction at address *a* in the executable code – is not executed, or is executed *n* times). A more formal description of the constraints could be more or less automatically taken into account by the tools. Second, many flow facts had to be determined manually (*eg*. loop bounds in the *memcpy* function according to the possible input values) and then specified manually to the WCET analyzer: this required much time for a single benchmark.

How we used OTAWA

To perform an automatic flow analysis, we used our oRANGE tool [10] which is not integrated to OTAWA at this time. oRANGE determines loop bounds from the source code. The flow facts related to other algorithmic structures as well as the loop bounds that could not be found automatically were specified as manual annotations.

Using the OTAWA framework, we have built a WCET analyzer that invokes the ARM binary code loader, the CFG builder, a flow fact loader that reads the flow facts provided by oRANGE as well as manual annotations, code processors that analyze the MAM (for instructions only) and the ARM7TDMI pipeline (they were specifically developed for the Challenge) and the IPET module. The MAM analysis is based on Abstract Interpretation [6] and the ARM7TDMI pipeline analysis uses execution graphs [34]. We reported results for the *m2t1* and *m2t3* MAM configuration. We considered that all the data were in the SRAM.

Comments on the WCET Tool Challenge

We found the Challenge very useful. First, it was one of our first experience with a "real" target (as part of our research activities, we are used to consider "generic" processor models). It also was the first time we really used our ARM loader. Second, the Challenge was an opportunity for people in the team to work for a common goal and to interact more deeply than usual.

5.4 RapiTime (written by G. Bernat and N. Merriam)

General comments

RapiTime observes and measures actual executions of the target program and therefore needs a suite of tests to be executed and observed. The "harness" module of the "debie1" benchmark contains a test suite, reached from the *main* function, that was created (by Tidorum Ltd) with two goals:

– To answer the analysis problems and questions defined for the "debie1" benchmark.

– To check that the modifications to the real DEBIE-1 flight software resulted in a benchmark program that still works as expected.

To satisfy the first goal, the test suite calls each “root” subprogram for each of the analysis problems defined for “debie1”, under several different conditions. The calls are grouped according to the input and environment constraints for each analysis problem so that the measurements for each problem can be identified from the whole mass of measurements.

To satisfy the second goal, the test suite makes some checks on the state of the “debie1” application, after executing each test. These checks are not part of the measured code.

For some target systems, RapiTime can use non-intrusive methods for observing and measuring executions. However, for the present case an intrusive method based on instrumentation was used. The RapiTime C-code instrumenter inserted instrumentation points in the “debie1” source-code. During execution, the instrumentation points emit trace information (using LPC2138 port P0) which is time-stamped and recorded by external equipment. After execution, the RapiTime analyzer parses the trace, relative to the known control-flow structure of the program and the locations of the instrumentation points, computes the execution-time distribution of each measured block of code, and computes a WCET estimate using all possible execution paths.

The test-suite code contains special instrumentation points that delimit the traces relevant to each analysis problem defined for “debie1”. Thus, while the benchmark program is only executed and measured once, the WCET analysis is done separately for each analysis problem. Moreover, RapiTime also computes coverage measures showing which parts of the code were executed.

The porting of RapiTime for the target was very simple, a single I/O port was used to trace the execution of the software using a logic analyser on an automatically instrumented version of the tests. The tests then were run and traces corresponding to each of the tests were collected and analysed successfuly.

We observed some rare but anomalously large execution times in the traces that indicate some transient behaviour of the processor. Only the *m2t3* configuration (MAM Mode 2, MAMTIM = 3) of the LPC2138 and “debie1” has been subjected to RapiTime analysis.

Even though RapiTime reports were generated, a lack of time to refine the analysis with annotations lead to unusable WCET results. The effort of completing the analysis and publishing the finished results will continue and will be reported in the next edition of the Challenge.

Comments on the WCET tool challenge

We believe that the Challenge is a very positive and that should be continued. This will strengthen the WCET community as a whole and unify the different research and development efforts.

5.5 TuBound (written by the TuBound group, see footnotes 5 and 6 on title page)

General remarks about the benchmarks and TuBound

At the current state of development, most effort went into the support of flow annotations and the transformation thereof. TuBound thus cannot yet cope with some of the additional constraints that were given in the problem descriptions.

This concerned mostly path annotations (*e.g.* function *f* will be executed at least once before the first invocation of function *g*) and variable value annotations, which caused overestimations in several benchmarks. For technical reasons, the interval analysis cannot yet accept user-supplied annotations. Work is underway to add support for this feature; however, it is not expected until after the deadline of the 2008 WCET Tool Challenge.

Although TuBound is conceived from ground up to be modular and to support multiple WCET analysis back ends, there was not enough time to port TuBound to the common ARM7 platform. We thus reported results for the C167 platform, which is a single-issue, in-order architecture with a jump cache.

About the WCET Tool Challenge

We found the WCET Tool Challenge a very good opportunity to evaluate the quality and performance of TuBound. It was a driver of many of TuBound's features to be implemented. The WCET Tool Challenge also revealed the necessity of supporting the proposed common target architecture to provide comparable results. We look forward to contributing to the next WCET Tool Challenge with a further enhanced version of TuBound. As a further improvement in comparability, we consider the availability of a highly expressive and widely supported common annotation language. Such a language has been demanded by our WCET'07 contribution on the “WCET Annotation Language Challenge” [20, 38], and a proposal on essential ingredients of such a language was contributed to the WCET'08 workshop [21]. A website has been created to collect contributions from the community to propose a common WCET annotation language [38].

5.6 Dortmund *wcc* (written by P. Lokuciejewski and D. Cordes)

Successes and suggestions for improvement

We considered only the flow-analysis problems, because we entered WCC'08 late (after the WCET 2008 workshop, in fact) and were pressed for time. Also, we saw no reason to enter WCET results for the current *wcc* target processor (TriCore) since no other WCC'08 participants use that target so no comparison would be possible.

The “debie1” benchmark is a complex real-world benchmark and its analysis is challenging for static analysis tools. However, its evaluation was a useful experience for us since it indicated which problems a static flow analysis must cope with in an industrial project. In the beginning, we had minor difficulties to figure out how some problems should be interpreted. However, with the intensive help of the organizers, all uncertainties could be removed and our static flow analysis succeeded in producing results for all flow analysis problems. To avoid interpretation problems for the future challenges, we are of the opinion that a common flow-fact annotation language supported by all participants that enables a formal description of the constraints would be beneficial.

The second class of benchmarks, the “rathijit” benchmarks, could be all successfully analyzed. Due to the typical flow fact questions about function execution counts, no understanding problems occurred. In general, the WCC'08 showed that our loop analyzer is suitable for the flow analysis of complex software and if future WCET Tool Challenges feature our target processor, we would be challenged to compare our WCET analysis results with other participants.

Comments on the WCET Tool Challenge

We consider the Challenge as a valuable activity. It motivated us to extend our static loop analysis by further functionalities of practical relevance. In contrast to the originally provided loop bound information, our analyzer is now able to bound the number of calls to functions. The evaluation of the “debie1” benchmark also indicated some cases where our static loop analyzer originally produced unnecessary overapproximations. Driven by this fact, a thorough review of our code entailed some modifications that improved our tool's analysis precision. Last but not least, the Challenge emphasizes the needs for a formal flow-fact language to enable a comparison of different tools.

6. Conclusion

How can we evaluate the success of WCC'08? Compared to the 2006 Challenge, the number of participating tools grew from five to six, but of these six, four are new tools that did not take part in 2006, and three of the 2006 tools did not take part in 2008. So there was a major change in the participant set. One reason for this may be that the organizers gave too much emphasis to the suggested common target (LPC2138); some potential participants abstained from WCC'08 because they did not have time to create ARM7 versions of their tools. The participation of MTime and RapiTime means that WCC'08 met its goal of including measurement-based tools. We also succeeded in defining pure flow-analysis problems unrelated to WCET analysis, but this did not yet attract the participation of pure flow-analysis tools. Even *wcc*, the only participant that answered only flow-analysis problems (and answered all of them), is a WCET-oriented tool. An important goal for the next Challenge should be to motivate the 2006 participants to rejoin the Challenge, without losing the new 2008 participants.

The number of benchmark programs was smaller in 2008 (5 benchmarks) than it was in 2006 (17 benchmarks, counting the two programs in PapaBench as two benchmarks). However, the number of analysis problems, or questions, was increased, because each WCC'08 benchmark poses several questions. Perhaps the next Challenge should not make such large changes in the benchmark set and in the nature of the benchmarks. It should perhaps reintroduce some of the benchmarks from 2006 (PapaBench in particular [24]). The suggestion from the MTime team to add a sequence of small, synthetic benchmarks, posing analysis problems of increasing difficulty, is also attractive.

Most WCC'08 participants found it difficult to understand the constraints on input values and execution flows defined for some of the WCC'08 analysis problems, in particular for the “debie1” benchmark. It is striking that the reports from many participants in section 5 ask for a more formal and standard way to define such things. This is certainly an issue for the organizers of the next Challenge, and also an opportunity to interact with the Annotation Language Challenge [20, 21].

Using a shared Wiki was successful. However, the next Challenge should strive to make the Wiki even more of a shared resource, jointly created and enjoyed by the Challenge participants, and indeed by any workers in the WCET analysis field.

With the conclusion of WCC'08, plans are being made for the next WCET Tool Challenge. The WCC'08 organizers suggest that the Challenge should be defined as a continuous process, allowing the addition of benchmarks, participants, and analysis results at any time, punctuated by an annual deadline. At the annual deadline, a snapshot of the results is taken and becomes the result of the Challenge for that year.

7. References

[1] AbsInt Angewandte Informatik GmbH, Worst-Case Execution Time Analyzer aiT for TriCore, 2008. http://www.absint.com/ait.

[2] Advanced RISC Machines, ARM7DMI Data Sheet. Document Number ARM DDI 0029E, Issue E, August 1995.

[3] ARTIST2 NoE – Cluster: Compilers and Timing Analysis, http://www.artist-embedded.org/artist/-Compilers-and-Timing-Analysis,45-.html.

[4] ARTIST2 Network of Excellence on Embedded Systems Design, http://www.artist-embedded.org/artist/.

[5] BERNAT, G., COLIN, A., and PETTERS, S.M., pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems, in: *Proceedings of the 3rd Int. Workshop on WCET Analysis* (WCET'2003), Porto, Portugal, 1 July 2003.

[6] BOURGADE, R., BALLABRIGA, C., CASSÉ, H., ROCHANGE, R., and SAINRAT, P., Accurate analysis of memory latencies for WCET estimation, in: *Int'l Conference on Real-Time and Network Systems* (RTNS), October 2008.

[7] CARLSSON, M., Worst Case Execution Time Analysis, Case Study on Interrupt Latency for the OSE Real-Time Operating System. Master's Thesis in Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden, 2002-03-18.

[8] CASSÉ, H., ROCHANGE, C., University Booth at DATE 2007 (Design, Automation and Test in Europe), April 2007.

[9] CORDES, D., Loop Analysis for a WCET-optimizing Compiler Based on Abstract Interpretation and Polylib (in German). Master's Thesis, Dortmund University of Technology, 2008.

[10] DE MICHIEL, M., BONENFANT, A., CASSÉ, H., and SAINRAT, P., Static loop bound analysis of C programs based on flow analysis and abstract interpretation, in: *IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications* (RTCSA), August 2008.

[11] European Space Agency, DEBIE – First Standard Space Debris Monitoring Instrument, http://gate.etamax.de/edid/publicaccess/debie1.php.

[12] FALK, H., LOKUCIEJEWSKI, P., and THEILING, H., Design of a WCET-Aware C Compiler, in: *Proceedings of the 4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia),* Seoul / Korea, October 2006.

[13] GUSTAFSSON, J., WCET Tool Challenge 2006, http://www.idt.mdh.se/personal/jgn/challenge/.

[14] GUSTAFSSON, J., The Worst Case Execution Time Tool Challenge 2006, in: S*econd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (ISoLA 2006), 2006.

[15] GUSTAFSSON, J., WCET Challenge 2006 – Technical Report, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-206/2007-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, http://www.mrtc.mdh.se/index.php?choice=publications&id=1209.

[16] HOLSTI, N., Computing Time as a Program Variable: A Way Around Infeasible Paths, in: *8th International Workshop on Worst-Case Execution Time Analysis* (WCET'2008), Prague, Czech Republic, July 1, 2008.

[17] HOLSTI, N., LÅNGBACKA, T., and SAARINEN, S., Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE Software, in: *Data Systems in Aerospace 2000* (DASIA 2000), EuroSpace and the European Space Agency, ESA SP-457.

[18] iSYSTEM AG, iF-DEV – iSYSTEM Free Tools for ARM7/ARM9/XScale, http://www.isystem.com/modules/news/article.php?storyid=25&location_id=0.

[19] iSYSTEM AG, Build 118 of iFDEV, http://www.isystem.si/SWUpdates/Setup_IFDEV_9_7_118/iFDEVSetup.exe.

[20] KIRNER, R., KNOOP, J., PRANTL, A., SCHORDAN, M. and WENZEL, I., WCET Analysis: The Annotation Language Challenge, in: *Post-Workshop Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis* (WCET 2007), (Pisa, Italy, July 3, 2007), 83 – 99.

[21] KIRNER, R., KADLEC, A., PUSCHNER, P., PRANTL, A., SCHORDAN, M., and KNOOP, J., Towards a Common WCET Annotation Language: Essential Ingredients. To appear in *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis* (WCET 2008), (Prague, Czech Republic, July 1, 2008).

[22] LOKUCIEJEWSKI, P., FALK, H., MARWEDEL, P., WCET-driven Cache-based Procedure Positioning Optimizations, in: Proceedings of the 20th Euromicro Conference on Real-Time-Systems (ECRTS 08), Prague, Czech Republic, July 2008.

[23] LOKUCIEJEWSKI, P., FALK, H., MARWEDEL, P., and THEILING, H., WCET-Driven, Code-Size Critical Procedure Cloning, in: *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems* (SCOPES), Munich, Germany, March 2008.

[24] NEMER, F., CASSÉ, H., SAINRAT, P., BAHSOUN, J.-P., and DE MICHIEL, M., PapaBench: a Free Real-Time Benchmark, in: *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis,* http://drops.dagstuhl.de/opus/volltexte/2006/678/.

[25] NXP Semiconductors, LPC2138 Errata Sheet, Version 1.8, July 9 2007, http://www.nxp.com/acrobat_download/erratasheets/ES_LPC2138_1.pdf.

[26] Mälardalen WCET benchmark collection, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[27] OLIMEX Ltd., LPC-H2138 Header Board for LPC2138 ARM7TDMI-S Microcontroller, http://www.olimex.com/dev/lpc-h2138.html.

[28] OTAWA website, http://www.otawa.fr/.

[29] Philips Semiconductors, LPC2138 User Manual, Rev. 01 — 24 June 2005, http://www.nxp.com/acrobat/usermanuals/UM10120_1.pdf.

[30] PRANTL, A., SCHORDAN, M., and KNOOP, J., TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis, in: *Proceedings 8th International Workshop on Worst-Case Execution Time Analysis* (WCET 2008), Prague, 2008.

[31] PUGH, W., et al., The Omega project: frameworks and algorithms for the analysis and transformation of scientific programs, University of Maryland, http://www.cs.umd.edu/projects/omega.

[32] Rapita Systems Ltd., RapiTime On Target Timing Analysis, http://www.rapitasystems.com/.

[33] RIEDER, B., PUSCHNER, P., and WENZEL, I., Using Model Checking to derive Loop bounds of general Loops within ANSI-C applications for measurement based WCET analysis, in: *Proceedings of the Sixth Workshop on Intelligent Solutions in Embedded Systems* (WISES'08), 2008, Regensburg, Germany.

[34] ROCHANGE, C., SAINRAT, P., A Context-Parameterized Model for Static Analysis of Execution Times, in: *Transactions on High-Performance Embedded Architectures and Compilers*, 2(3), Springer, October 2007.

[35] SWEET website, http://www.mrtc.mdh.se/projects/wcet/sweet.html.

[36] TAN, L., The Worst Case Execution Time Tool Challenge 2006: The External Test, in: S*econd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (ISoLA 2006), 2006.

[37] Tidorum Ltd., Bound-T website, http://www.bound-t.com/.

[38] WCET Analysis: The Annotation Language Challenge website, CoSTA project, http://costa.tuwien.ac.at/languages.html.

[39] WCET Tool Challenge 2008, http://www.mrtc.mdh.se/projects/WCC08/.

[40] WCET Tool Challenge 2008 Wiki, http://www.mrtc.mdh.se/projects/WCC08/doku.php?id=start. Read-only public access. For editing access, contact the WCC'08 steering group.

[41] WENZEL, I., KIRNER, R., RIEDER, B., and PUSCHNER, P., Measurement-Based Timing Analysis, in: *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (ISoLA'08), to appear, 2008, Porto Sani, Greece.

[42] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., and STENSTRÖM, P., The worst-case execution time problem — overview of methods and survey of tools, in: ACM Transactions on Embedded Computing Systems, Volume 7, Issue 3, April 2008.

A TOOL FOR AVERAGE AND WORST-CASE EXECUTION TIME ANALYSIS

David Hickey[1], Diarmuid Early[1] and Michel Schellekens[1]

Abstract
We have developed a new programming paradigm which, for conforming programs, allows the average-case execution time (ACET) to be obtained automatically by a static analysis. This is achieved by tracking the data structures and their distributions that will exist during all possible executions of a program. This new programming paradigm is called $\mathcal{MOQA}$ and the tool which performs the static analysis is called Distritrack. *In this paper we give an overview of both $\mathcal{MOQA}$ and Distritrack. We then discuss the possibility of extending Distritrack for static worst-case execution time (WCET) analysis of $\mathcal{MOQA}$ programs using the tight tracking of data structures already being performed.*

1. Introduction

Much work is being done on the development of ways to predict program execution times. The efforts are concentrated into two areas - *complexity theory* in which various time measures have been developed for counting the basic number of operations in a program and *real-time systems* in which constraints on the execution times of programs are imposed, e.g. deadlines, cost, etc.

In general however, the static analysis of programs to determine any property, one of which is time, is known to be very difficult in practice. Measuring ACETs automatically is no different. Some analysis techniques have been developed, e.g. [2], but these tend to be quite complicated involving many difficult mathematical techniques. Along with this, it is required that in some cases the algorithms are programmed in an unfamiliar style when compared to general programming languages.

$\mathcal{MOQA}$ involves a way to determine statically the distribution of all possible data structures at any point in a program. This makes an ACET analysis possible. The underlying mathematical techniques are less complicated than previous approaches and allow a common programming style. Currently $\mathcal{MOQA}$ programs are coded in Java using an API implementing $\mathcal{MOQA}$'s operations.

Distritrack is the tool that has been developed to automate the static ACET analysis of $\mathcal{MOQA}$ programs. It combines elements of a number of static analysis techniques in order to track the data structures and their distributions as set out in $\mathcal{MOQA}$. The output of an analysis is generally a recurrence equation representing the number of basic operations, e.g. comparisons, swaps or Java bytecode instructions, executed on the data structures. As future research, low-level timing information for specific hardware could be combined with this in order to determine the expected "real" ACET (i.e. clock cycles, milliseconds, etc.) for the program being considered taking into account caching, pipelining, etc.

ACETs can be used in conjunction with other execution time measures in soft real-time systems to

[1]Centre for Efficiency Oriented Languages, Department of Computer Science, National University of Ireland, Cork

estimate deadlines. While deadlines determined in this way only guarantee enough time for a majority of their associated tasks, they may however lead to a significant improvement in the utilisation of system resources [7]. When deadlines are hard, WCET is a more suitable execution time measure. Like ACET, this is often difficult to obtain. Here we examine if the tight tracking of data structures that is performed by Distritrack might in fact also facilitate a WCET analysis.

This paper is organised as follows. In Section 2. $\mathcal{MOQA}$ is introduced. Then in Section 3. an overview of Distritrack is given along with details of how data structures are tracked. Section 4. gives an example of a $\mathcal{MOQA}$ program and the corresponding ACET output by Distritrack. Next in Section 5. we examine possible ways of extending Distritrack's current analysis in order to obtain WCET estimates. Finally in Section 6. some concluding remarks are given.

2. $\mathcal{MOQA}$

$\mathcal{MOQA}$ [9, 10] is a special purpose high-level language for data (re)structuring. Its data structures are simply specified as finite partial orders. Its operations are based on the classical abstract data type operations. However, each operation has been purposely designed to guarantee that data collections remain random throughout the computations. This in turn guarantees a modular ACET analysis.

In this section an overview of $\mathcal{MOQA}$ is given based on the main ideas discussed in [9].

2.1. Data Structures

The basic data structure in the current implementation of $\mathcal{MOQA}$ is a series-parallel partial order (SPPO) which is a partial order that only allows nodes to be in series, denoted by $\otimes$, or in parallel, denoted by $\|$. For example the SPPO in Figure 2(a) can be represented in series-parallel notation by $d \otimes ((b \otimes a) \| c)$. Sub-structures, which can be single nodes or more complex SPPOs, are called *components*.

The data of the language are labellings of the data structures. A data-labelling is simply an assignment of a finite number of values to each node of the data structure so that the directed links of the data structure are respected, i.e. if there is a directed link from a node x to a node y, then the label assigned to x must be less than the label assigned to y. These labels can be any value, e.g. natural or real numbers, words, other data structures containing data such as trees, etc. Any two labels need to be comparable with respect to a given order on labels. For instance, the order on natural number labels typically is the usual order on natural numbers.

$\mathcal{MOQA}$ programs compute over data-labellings, and will at each stage transform data-labellings to new data-labellings. In such computations it is important to identify the states that data-labellings can be in.

A *state* represents a collection of order-isomorphic data-labellings, i.e. data-labellings whose labels are arranged in the same relative order within data structures.

We illustrate this with the data-labellings for the tree of size 4 given in Figure 1. If we use four distinct values, say a, b, c, d, to represent the states of the same tree, where $a < b < c < d$, we have only three possible states as displayed in Figure 2. Note that the data-labelling in Figure 1(a) matches the state in Figure 2(a) and data-labelling in Figure 1(b) matches the state in Figure 2(b).

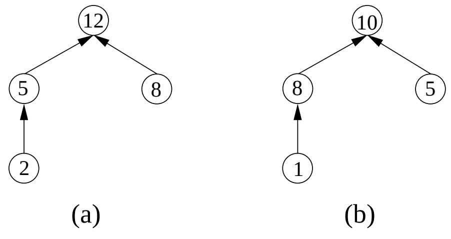


Figure 1. Data-labellings on data structures.

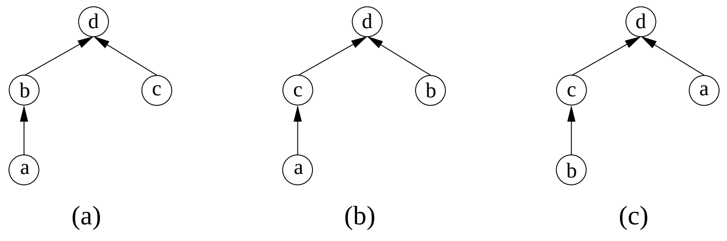


Figure 2. Data-structure states.

Essentially, states reflect the relative order that the labels can be in, on any given data structure. The values of the labels are irrelevant in this context, only their relative order is captured.

For any given data structure, the finite collection of the set of states over this data structure, is referred to as the *random structure* over the given data structure.

This amounts to the assumption that inputs for software are equally likely to occur in any of a given number of finite states. Random data can be concisely captured as above via the notion of a random structure. In practice of course, there may be several possible data structures. To represent this, the notion of random bags is introduced. A *random bag* consists of finitely many random structures, $R_1, \ldots, R_n$, each of which has a *multiplicity* K_i, where $i \in 1, \ldots, n$, which is a natural number used in the calculation of the probabilities involved in the distribution.

2.2. Operations

Operations in $\mathcal{MOQA}$ map input random bags to output random bags. Operations which correctly do this are *random bag preserving*.

The multiplicities of the input random bags are the key to the calculation of the ACETs. The ACET for an operation P with input random bag $R = \{(R_1, K_1), \ldots, (R_n, K_n)\}$ is

$$\overline{T}_P(R) = \sum_{i=1}^{n} \frac{K_i|R_i|}{|R|} \times \overline{T}_P(R_i) \tag{1}$$

where $|R_i|$ indicates the number of states in R_i, $\frac{K_i|R_i|}{|R|}$ is the probability of R_i occurring and $\overline{T}_P(R_i)$ is the ACET of executing P with input R_i.

Then, taking random bag preserving programs/operations P and Q such that executing P on random bag R results in random bag R', the ACET of executing P followed by Q is:

$$\overline{T}_{P;Q}(R) = \overline{T}_P(R) + \overline{T}_Q(R') \tag{2}$$

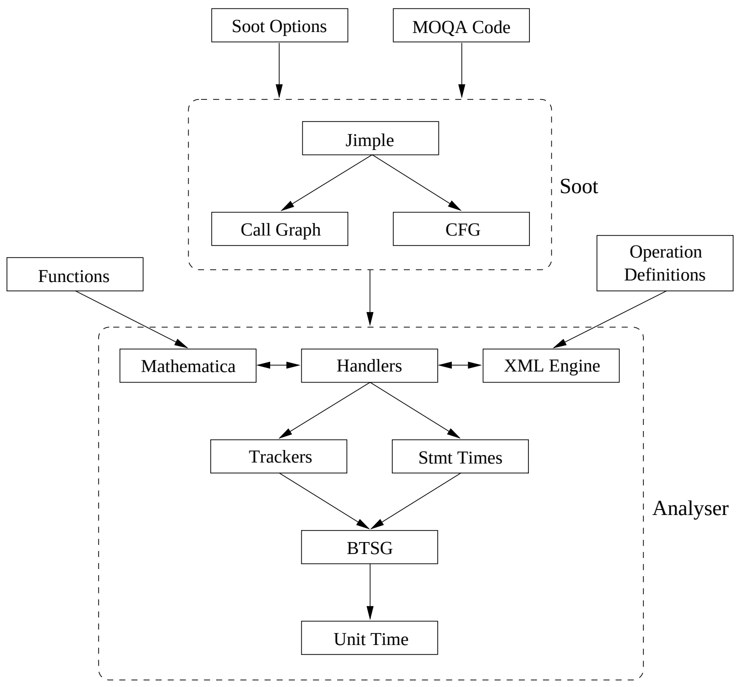


Figure 3. Distritrack architecture.

For the purpose of this paper we will focus on two main $\mathcal{MOQA}$ operations. For a complete description of the operations, designed to capture traditional data structuring operations in a randomness preserving fashion, we refer the reader to the Springer book [9].

Here we focus on the $\mathcal{MOQA}$ deletion operation $\overline{Del}$, the $\mathcal{MOQA}$ product operation $\bigotimes$ and the $\mathcal{MOQA}$ split operation.The $\mathcal{MOQA}$ product operation enables the user to "merge" two $\mathcal{MOQA}$ data structures into a new $\mathcal{MOQA}$ data structure. For the specific case of list labellings, this operation corresponds to the traditional merging of two lists. The product of a single element data structure with a larger data structure amounts to the classial insertion operation of inserting an element into a given data structure. The $\mathcal{MOQA}$ product operation uses the traditional PushUp and PushDown operations on labels as part of its internal working. The $\mathcal{MOQA}$ deletion operation enables the user to remove a label from a given labelling in such a fashion that the original data structure is reduced to a random bag of new data structures, with labellings not containing the deleted label. The $\mathcal{MOQA}$ split operations simply reorders label relative to a given label, similar to the partitioning operation of standard Quicksort. Details for these operations are provided in [9]. Most applications of the operations reduce to the specific cases outlined above and hence the application of the operations in practice are a great deal simpler than the definitions over general random structures as presented in [9]. We will indicate later on how to handle the worst-case analysis for the case of these two operations.

3. Distritrack

3.1. Overview

Figure 3 gives an overview of the design of Distritrack.

The most important aspect of Distritrack [3] is its ability to track the $\mathcal{MOQA}$ data structures. This is the fundamental requirement in $\mathcal{MOQA}$ which allows the ACET of its operations to be calculated. To allow this, *data structure representations* were formulated. These reflect the series-parallel nature of $\mathcal{MOQA}$'s data structures and facilitate the application of the *composition laws* (cf. Chapter 6 of [9]) and the evaluation of formulae for multiplicities and the numbers of states.

At any point in a program each variable referencing a $\mathcal{MOQA}$ data structure has a *random bag tracker* associated with it. A random bag is represented as a collection of the data structure representations, in effect corresponding to random structures, which together represent all possible states of the corresponding data structures.

To achieve this, the static analysis performed by Distritrack takes each statement in the code and *handlers* simulate its effects on the actual data structures by altering the corresponding data structure representations in the random bags being tracked. This can be viewed as an abstract interpretation of the semantics of $\mathcal{MOQA}$ operations. The "abstract" semantics are encapsulated in XML and processed by Distritrack's XML engine.

In order to be able to analyse the code effectively on a statement by statement basis, Distritrack performs a flow analysis [5] of the program. This is done by the construction of a control-flow graph and call graph for the program using a tool called Soot [8]. Information on the analysis is encapsulated in another graph called a BTSG.

Distritrack gives special attention to statements such as `for` loops[2], recursive calls and `if` statements which affect the control flow. The first two complicate the tracking of the data structure representations because the effects of a statement can not generally be analysed in isolation and have to be simulated for a symbolic number of executions, e.g. n. To solve this problem the use of *recursive data structures* (RDS) [4] was incorporated.

RDSs are especially suited for recursion. In $\mathcal{MOQA}$ theory there are two templates defined for recursion based on the series-parallel nature of the data structures. For Distritrack these have been generalised to give a more standard template for recursion. It is called *series-parallel recursion* and is defined informally as follows:

$$Q(Y) = R(Y); P(Q(Y_1); \ldots ; Q(Y_k))$$

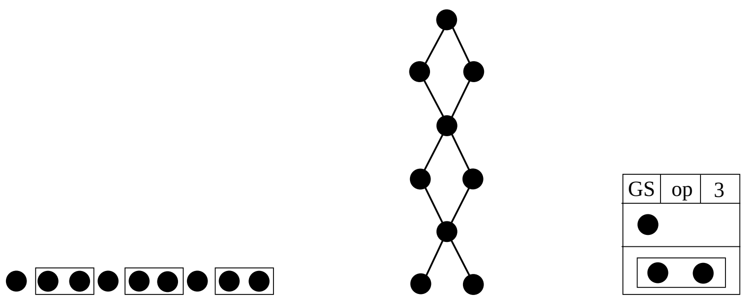
where R transforms SPPO Y into $Y_1, \ldots, Y_k$ which can be in series or parallel and P processes the results of the recursive calls. Either P or R can be optionally excluded.

We also developed a set of rules for the construction of RDS definitions directly from the code associated with `for` loops. These rules are quite powerful but also require templates to be imposed on the loop bodies.

Processing `if` statements and more specifically the conditions they depend on was very challenging. However for a limited category of conditions calculating probabilities and determining the effects on data structure representations is possible to automate in Distritrack. When a probability is not possible to calculate *cases* are incorporated into the formulae for the ACETs, effectively resulting in separate ACETs for true and false branches.

Evaluating the formulae at various points in the analysis is achieved by interacting with Mathematica.

[2]Other types of loops like `while` are currently not considered.



(a) $op = \|$ (b) $op = \otimes$ (c)

Figure 4. Group structure with a repeat value set to 3.

The final output of Distritrack are ACET formulae built in a modular way from the program statements and RDS formulae if required. These will generally be recurrence equations.

The analysis performed by Distritrack is quite flexible. The analysis itself requires little user interaction with only some guidance on the processing of RDSs being provided through code annotations. The tool can handle not only all the features of $\mathcal{MOQA}$ but also many aspects of the Java programming language. The analysis is interprocedural and can span multiple Java classes. Constructors, overloaded methods, class hierarchies and many other complex features can be handled.

Here we give an overview of the data structure representations that are incorporated into Distritrack for tracking the random bags. We will also discuss how some of the required information for calculating ACETs can be derived from these representations.

3.2. Data Structure Representations

The means by which Distritrack tracks values during its analysis of $\mathcal{MOQA}$ programs is through the use of *trackers*. The most important of these are *random bag trackers*. A random bag tracker is a set of *random structure representations* built using *fundamental SPPO (series-parallel partial order) representations*. A multiplicity is attached to each random structure representation.

Currently the fundamental SPPO representations incorporated into Distritrack are *empty structure*, *basic structure*, *primitive structure* and *group structure*. An empty structure contains nothing and a basic structure represents a single data structure node. A primitive structure contains exactly two components directly reflecting the binary nature of the series and parallel operators. Group structures are n-ary structures, i.e. they can contain an arbitrary number of components, all joined either in series or in parallel. The components within the group structure can have a *repeat value* set which determines the number of occurrences of the components defined in the group structure. Figure 4 shows an example of this. The repeat value can also be set to a symbolic value. Thus group structures form an important aspect of Distritrack's symbolic analysis.

In practice the tracking of the data structures is quite complicated and requires some more sophisticated representations, including RDSs as mentioned above.

The tracking of data structures can be compared to *shape-analysis* [11]. For example, the data structure representations can be viewed as *shape graphs* and the use of symbolic values for repeat values can be viewed as *summarization*. In both cases an abstract interpretation of the operations that modify the shape graphs is used.

3.3. Calculating Operation ACETs

For a SPPO representation the most important values that need to be obtained are summarised as follows:

$|s|$ The size of the entire SPPO s.

$|M(s)|$ The size of the set of maximal nodes (informally defined as having no parents) in s.

$|m(s)|$ The size of the set of minimal nodes (informally defined as having no children) in s.

$L(s)$ The number of states possible on s.

Composition Laws The ACET to manipulate labels in an SPPO based on the series-parallel structure. Currently the ACET is defined as the number of comparisons required, which is suitable for the data restructuring algorithms currently implemented in $\mathcal{MOQA}$. For example the simple composition law σ_{up}, which gives the average number of comparisons to push the minimum label from a minimal node up to a maximal node, is defined as follows ($\bullet$ is a single node):

$$\begin{aligned}\sigma_{up}(\bullet) &= 0\\ \sigma_{up}(A \otimes B) &= \sigma_{up}(A) + \sigma_{up}(B) + |m(A)|\\ \sigma_{up}(A \| B) &= \frac{|A|\sigma_{up}(A) + |B|\sigma_{up}(B)}{|A| + |B|}\end{aligned}$$

Multiplicity The multiplicity of the random structure.

Many of these will be derived in a recursive way, with empty and basic structures providing the base cases. The multiplicities are determined by the definitions of operation behaviour in the abstract semantics supplied to Distritrack.

As an example lets look at primitive structures. A primitive structure can be represented as follows: $ps = s_1$ *op* s_2, where *op* is either $\otimes$ or $\|$ and s_1, s_2 are two components. The following lists the recursive way in which the size related values are obtained:

- $|ps| = |s_1| + |s_2|$
- If *op* is $\otimes$ then $|M(ps)| = |M(s_1)|$. If *op* is $\|$ then $|M(ps)| = |M(s_1)| + |M(s_2)|$.
- If *op* is $\otimes$ then $|m(ps)| = |m(s_2)|$. If *op* is $\|$ then $|m(ps)| = |m(s_1)| + |m(s_2)|$.

Functions for counting the number of states and the composition laws are binary operations based on the series parallel nature of the data structures. Therefore they can very naturally be applied to primitive structures.

Though more complicated to derive, these values can also be obtained for group structures.

With these formulae, the ACET for an operation can then be obtained using Equation 1 where the ACET on each random structure is derived using the composition laws.

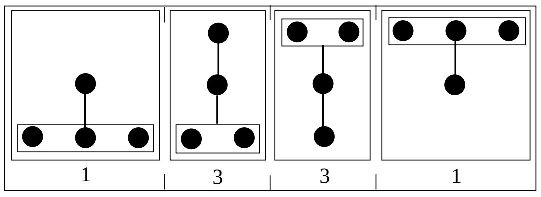


Figure 5. Output of $\mathcal{MOQA}$'s split operation on an atomic random structure of size 4.

4. Example

Listing 1 gives the code for Quicksort implemented in $\mathcal{MOQA}$. For simplicity the code for Java 5's generics is omitted. The input to method1 is considered always to be a list.

Line 9 is an annotation which tells Distritrack to build a RDS definition for the first of the method's parameters. In this case the RDS is a a simple single random structure - the sorted list. The representation for this is built into Distritrack and is called *Linear*. In general however Distritrack will generate a new RDS definition based on the code of the recursive method. All annotations to Distritrack are optional and, except for those related to generating RDSs, give information to Distritrack which may make the output simpler or the analysis more efficient.

Line 13 contains the $\mathcal{MOQA}$ operation split which partitions the input list around a random pivot. Figure 5 shows the output random bag of split for an input list with 4 nodes. The number under each random structure represents its multiplicity. In practice Distritrack tracks the output for symbolic sizes.

Lines 15 and 16 then recurse on the resulting partitions similar to the non-$\mathcal{MOQA}$ version of Quicksort code.

Listing 1. Quicksort in MOQA.

```
 1 public class QuicksortTest {
 2
 3    public OrderedCollection method(
 4           OrderedCollection oc) {
 5        quicksort(oc);
 6        return oc;
 7    }
 8
 9    @Transform(param=0, rep=RDSBuild.SR, name=''Linear'')
10    private void quicksort(OrderedCollection oc) {
11        if(oc.size() > 1) {
12            NodeInfo partitionNI = oc.getDirectNodeInfoIter().next();
13            OrderedCollection partition = oc.split(partitionNI);
14            Iterator aboveAndBelow = partition.getDirectSubsetIter();
15            quicksort(aboveAndBelow.next());
16            quicksort(aboveAndBelow.next());
17        }
18    }
19 }
```

Listing 2. Quicksort ACET Mathematica package.

```
quicksort[n1_] := Which[Greater[n1,1], Plus[-1,n1,
    Sum[Times[Power[n1,-1], quicksort[Plus[-1,n1,Times[-1,r0]]]], {r0,0,Plus[-1,n1]}],
    Sum[Times[Power[n1,-1], quicksort[r0]], {r0,0,Plus[-1,n1]}]],
    True,0];
```

```
method[n0_] := quicksort[n0];
```

Listing 2 gives the Mathematica package generated by Distritrack for the ACETs of the methods analysed in the Quicksort example. The ACET of the `quicksort` method corresponds to the standard Quicksort recurrence:

$$qs[n] = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} qs[i]$$

5. Extending Distritrack for WCET Analysis

Adapting Distritrack for a WCET analysis requires new composition laws which calculate the worst-case number of basic operations executed on a random structure when a $\mathcal{MOQA}$ operation is encountered. Effectively these will select one of the states represented within the random structure which gives the largest execution time.

To illustrate how this can be achieved we briefly discuss the worst-case execution times for the two main operations discussed in the present paper: the product operation $\bigotimes$ and the deletion operation $\overline{Del}$.

5.1. Worst Case Running Times of Basic Operations

5.1.1. Delete

Let R be a random structure with an underlying partial order A. If we call $\overline{Del}(r, k)$ on each labeled SPPO r in R, the greatest number of comparisons made by any operation call is $\delta_{up}^{W}(A, k)$.

The δ_{up}^{W} function satisfies the following series-parallel recurrences (where A and B are non-empty, disjoint partial orders):

1. $\delta_{up}^{W}(A \otimes B, k) = \begin{cases} \delta_{up}^{W}(A, k) + |B_{min}| - 1 + \delta_{up}^{W}(B, 1) & k \leq |A| \\ \delta_{up}^{W}(B, k - |A|) & k > |A| \end{cases}$
2. $\delta_{up}^{W}(A \| B, k) = \max\left(\delta_{up}^{W}(A, \min(k, |A|)), \delta_{up}^{W}(B, \min(k, |B|)\right)$
3. $\delta_{up}^{W}(\bullet, k) = 0$

5.1.2. Product

Let R be a random structure with an underlying partial order A. If we replace the smallest label on each labeled SPPO in R with a label which is larger than k members of the label set and smaller than the others, and then call PushUp on the node with that label which simply pushes up the label, the greatest number of comparisons made by any operation call is $\pi_{up}^{W}(A, k)$. We define $\pi_{down}^{W}(A, k)$ in a similar manner by replacing the *largest* label and calling Push*Down* to push down a label.

If we similarly replace the smallest label on each labeled SPPO in R with a label greater than k members of the label set and call a PushUp, the new label may be the label of a maximal node in the output labeled SPPO. If this happens, then the greatest number of comparisons made by any PushUp operation in these cases is $\mu_{up}^{W}(A,k)$. If not, then $\mu_{up}^{W}(A,k) = -\infty$. We define $\mu_{down}^{W}(A,k)$ in a similar manner by replacing the *largest* label and calling Push*Down*.

Let $T_P^W[I_1, I_2]$ be the worst-case running time for the unary product on the components I_1 and I_2 over all labellings in the random structure R with underlying structure $I_1 \| I_2$. Then we have

$$\begin{aligned} T_P^W[I_1, I_2] \leq \; & (\min(|I_1|, |I_2| + 1))(|I_{1,max}| + |I_{2,min}| - 1) + \\ & + \sum_{i=1}^{\min(|I_1|,|I_2|)} \left[\pi_{down}^{W}(A,i) + \pi_{up}^{W}(B, |B| + 1 - i)\right] \end{aligned}$$

The π_{up}^{W} and μ_{up}^{W} functions satisfy the following series-parallel recurrences (where A and B are non-empty, disjoint partial orders):

1. $\pi_{up}^{W}(A \otimes B, i) = \begin{cases} \max\left(\pi_{up}^{W}(A,i), \mu_{up}^{W}(A,i) + |B_{min}|\right) & i \leq |A| \\ \\ \pi_{up}^{W}(A, |A|) + |B_{min}| + \pi_{up}^{W}(B, i - |A|) & i > |A| \end{cases}$
2. $\pi_{up}^{W}(A \| B, i) = \max\left(\pi_{up}^{W}(A, \min(i, |A|)), \pi_{up}^{W}(B, \min(i, |B|))\right)$
3. $\mu_{up}^{W}(A \otimes B, i) = \begin{cases} -\infty & i \leq |A| \\ \mu_{up}^{W}(A, |A|) + |B_{min}| + \mu_{up}^{W}(B, i - |A|) & i > |A| \end{cases}$
4. $\mu_{up}^{W}(A \| B, i) = \max\left(\mu_{up}^{W}(A, \min(i, |A|), \mu_{up}^{W}(B, \min(i, |B|)\right)$
5. $\pi_{up}^{W}(\bullet) = \mu_{up}^{W}(\bullet) = 0$

5.2. Extending Distritrack

The advantage of extending Distritrack for WCET analysis is that the input-output trace that it currently undertakes leads to very accurate WCETs. With the new composition laws Distritrack can compute the WCET for each random structure representation in a random bag being tracked as input into an operation. This can be done without altering the way in which the random bag trackers are generated. When an entire method/program is analysed, the information obtained for each operation is combined and the sequence of operation WCETs for the overall WCET can be derived. Existing WCET tools already incorporate techniques for finding the maximum time required for different execution paths in a program, for example [1, 6]. These techniques could be applied in a similar fashion to determine the WCET from the times associated with the random structures.

As an alternative, Distritrack could maintain the WCET to build a random structure up to each point in the program analysis. Say operation op_i is being analysed and its input is the random bag R. Let $WCET_{i-1}(R_j)$ be the WCET required to build random structure R_j within R by the $i-1$ operations before op_i and $T_i^W(R_j)$ be the WCET of executing op_i on R_j. Each random structure in the output random bag resulting from the execution of op_i on R_j will be associated with the WCET $WCET_{i-1}(R_j) + T_i^W(R_j)$.

Then, after the last operation in a path of execution in a program, the WCET of that path will be the maximum WCET value from the random bag output from the operation.

Other than this, the static analysis currently performed by Distritrack can remain largely unchanged.

This however does not use all the information provided by the data structure representations built by Distritrack. The multiplicities may sometimes be useful in obtaining time estimates for the inputs to an operation which are “most likely” to occur. In [7] this is shown to be important when, using the WCET alone to determine deadlines in a real-time system, there is a large waste of resources when the input that causes it occurs very infrequently. Therefore Distritrack can, for example, drop a WCET value if the WCET occurs in a random structure that has a probability of occurring less than $\frac{1}{20}$. This of course is only relevant for soft real-time systems.

However multiplicities in this case only lead to a partial solution. While the states within a random structure have a uniform distribution, we currently do not have information on the distribution of the execution times over the states. It may be possible however to develop ways of extracting such information, again similar to the way the current composition laws derive ACETs.

Multiplicities may also indicate which random structures contain the worst case state for an operation. It has been observed that the WCET generally occurs in a random structure which contains the largest *atomic* (single nodes in parallel) components. In terms of *divide and conquer*, this makes sense. As it turns out, the output from operations that create atomic components in series appear to always have the lowest multiplicity attached to the random structure containing the largest atomic components. This may be because these random structures contain more states and therefore fewer copies occur. An example of this can be found in the output of $\mathcal{MOQA}$’s split operation. For size 4, in Figure 5 it can be clearly seen that the random structures containing the largest atomic components have the lowest multiplicity. These also form the worst case input into the recursive call of quick sort.

6. Conclusion

In this paper we have given an overview of a new programming paradigm called $\mathcal{MOQA}$ and a corresponding tool called Distritrack. Distritrack performs a static analysis of $\mathcal{MOQA}$ programs, tracking the data structures and their distributions as they are input to and output from program statements in order to derive the ACET.

We discussed how Distritrack can have its ACET static analysis extended to derive WCET and other time measures using the information provided by the tracking of the data structures. We have provided some initial ideas to serve as a basis on which to investigate this further, supporting a future implementation of the tool in which the estimates provided for improved resource budgeting in soft real-time applications can be supplemented with accurate WCET deadlines for hard real-time applications.

The price of the accurate results obtained by Distritrack are some limitations on the static analysis which are required to obtain the tight tracking of the data structures, e.g. the analysis must be context-sensitive in that all program paths have to be analysed separately.

References

[1] R. Chapman, A. Wellings, and A. Burns. Integrated program proof and worst-case timing analysis of spark ada. In *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[2] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theor. Comput. Sci.*, 79(1):37–109, 1991.

[3] David Hickey. Distritrack: Automated average-case analysis. In *QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 213–214, Washington, DC, USA, 2007. IEEE Computer Society.

[4] C. A. R. Hoare. Recursive data structures. *International Journal of Parallel Programming*, 4(2):105–132, 1975.

[5] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[6] Peter Puschner. Worst-case execution time analysis at low cost. *Control Engineering Practice*, 6:129–135, Jan. 1998.

[7] Peter Puschner and Alan Burns. Time-constrained sorting – a comparison of different sorting algorithms. In *Proc. 11th Euromicro International Conference on Real-Time Systems*, pages 78–85, Jun. 1999.

[8] McGill University Sable. Soot, a java optimization framework. www.sable.mcgill.ca/soot.

[9] M. P. Schellekens. *A Modular Calculus for the Average Cost of Data Structuring*. Springer, August 2008. http://www.springer.com/computer/foundations/book/978-0-387-73383-8.

[10] M. P. Schellekens $\mathcal{MOQA}$*; unlocking the potential of compositional static average-case analysis.* In *Journal of Logic and Algebraic Programming*, accepted for publication, to appear.

[11] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 1–17, London, UK, 2000. Springer-Verlag.

APPLYING WCET ANALYSIS AT ARCHITECTURAL LEVEL

Olivier GILLES, Jérôme HUGUES[1]

Abstract
Real-Time embedded systems must enforce strict timing constraints. In this context, achieving precise Worst Case Execution Time is a prerequisite to apply scheduling analysis and verify system viability. WCET analysis is usually a complex and time-consuming activity. It becomes increasingly complex when one also considers code generation strategies from high-level models.

In this paper, we present an experiment made on the coupling of the WCET analysis tool Bound-T and our AADL to code generator OCARINA. *We list the different steps to successfully apply WCET analysis directly from model, to limit user intervention.*

1. Introduction

Distributed Real-time and Embedded (DRE) systems must enforce strict timing constraints. Run-time mechanisms exist to control the execution time of each processing thread, and eventually detect overrun, like Ada 2005 execution time timers. Yet, these techniques are usually resource consuming, and not sufficient when building critical systems [11].

A better option for resource-constrained or critical systems is to rely on precise WCET analysis techniques. These techniques rely on the careful examination of the source code and/or assembly-level code to extract longest execution path. However, they are hard to master, and time-consuming. Yet, one needs evaluation of the WCET of some functions early in the definition of the software system.

At the same time, model-based development, the idea that a system can be described in a high-level formalism and then leads to fully generated systems emerge. We claim that this approach is interesting provided that the modeling process fully integrates engineering concerns for real-time systems, including performance analysis.

In this paper, we present a process for evaluating DRE systems WCET metrics relying on both architectural model and binary code analysis. In the second section, we present a taxonomy of relevant information for evaluating a system's performance and how one can model the whole system and its properties. In the third section, we present a full process to analyse models performance in an integrated framework. In the last section, we provide a use case, showing how to achieve full system evaluation and conclude on the needs for more advanced analysis techniques.

[1]GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
`olivier.gilles@enst.fr`, `jerome.hugues@enst.fr`
This work is funded in part by the Flex-eWare and MOSIC projects

2. Defining a toolchain for building critical real-time systems

In this section we list contextual information on our research work, prior to illustrating the benefits of automating WCET analysis in an automated toolchain.

2.1. A taxonomy of performance criteria

DRE systems must comply to multiple and contradictory constraints. In this section, we present a taxonomy of those constraints and then classify them relatively to their analyzability.

Deterministic behavior : Ensuring a fully deterministic behavior is a complex issue. As a general rule, one should select a computational model that is amenable to full analysis. This usually implies restricting the set of constructs allowed at either model or programming levels.

Schedulability : Once deterministic behavior is achieved, one can apply scheduling analysis on the set of tasks using scheduling-related data (eg. periodicity, deadline, execution time) of each thread in the software model. Furthermore, interferences of critical section accesses must be analyzed too, and thus expressed in the model.

WCET : this is the key factor for determining schedulabity of a system. Worst-Case Execution Time provides a hint on the duration of some computations. A wrong estimate leads either to pessimistic usage of resource, or some runtime errors. While direct analysis of the binary code can be used in order to extract some values (such as a subprogram WCET), more complete evaluations will need to capture the system semantics. Architectural-level relations, in particular, must be captured in order to make scheduling and latency analysis easier. (cf. [6] and [12]). Still, the analysis of the WCET should be also compatible with both the modeled system, and the code that will actually execute it.

Therefore, we claim that these constraints are better expressed at model-level, and then enforced in an automatic code generation and then analysis process process. Code generation process would enforce deterministic behavior and also ensure generated code is amenable to WCET analysis at object or source code level. We selected the AADL, an architectural description language, to describe framework-level and program-level properties. We then present the code generator we developed, and show how to combine it with the WCET analysis tool Bound-T [7].

2.2. AADL

AADL (*Architecture Analysis and Description Language*) [8] aims at describing DRE systems by assembling components. AADL allows for the description of software and hardware parts. It focuses on the definition of interfaces, and separates the implementations from these interfaces.

An AADL description is made of *components*. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`). Components describe well identified elements of the actual architecture. *Subprograms* model procedures as in C or Ada. *Threads* model the active part of an application (such as POSIX threads). AADL threads may have multiple operational modes. Each mode may describe a different behavior and property values for the thread. *Processes* are memory spaces that contain the *threads*. *Processors* model micro-processors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, *buses* model all kinds of networks, wires, *devices* model sensors, etc.

An AADL model also describes non-functional facets: embedded or real-time characteristics of the components (execution time, memory footprint...), behavioral descriptions, etc. Description can be extended either through new property sets, or through annexes. Annexes are extensions to the core language. A complete introduction to the AADL can be found in [2].

We have developed the OCARINA [13] tool-suite to manipulate AADL models. OCARINA proposes AADL model manipulation based on a compiler-like API. "Back-end" modules can generate formal models, perform scheduling analysis and generate distributed high-integrity applications in Ada. Generated code relies on the POLYORB-HI middleware to ensure communications and task allocation. POLYORB-HI ensures that a minimal and reliable middleware is generated for a given distributed application. Furthermore, it relies on strong design patterns to ensure the code is compatible with the requirements from the High-Integrity domain, easing further analysis. For a complete description of the code generation strategy enforced by POLYORB-HI, please refer to [3].

2.3. Setting framework properties

As explained above, AADL not only offers a language to describe architectural relations, but also allows to define run-time oriented properties such as a subprogram WCET, the worst-case duration of a context switch, etc. However, the user must provide those values, which is definitively not a trivial task, since such values are highly architecture-dependent and often non-deterministic.

To our knowledge, no reliable method exists to compute such values from high-level architectural level. In order to obtain those values, we proceeded to an analysis of the binary code generated from the architectural model by OCARINA, using tools such as Bound-T, which allows to extract a local subprogram upper bound on the WCET in terms of processor cycles.

3. Evaluation tool suite

A design-level software model cannot directly compute all of system properties, as some of them are highly OS-specific, hardware-specific or compiler-specific. In particular, subprogram WCET or code size cannot be guessed reliably at design level. In order to perform a realistic evaluation, one should first generate the exact binary code for the system, and then perform analysis on the code and then using together the software architecture and the binary code metrics to refine the initial model. In this section, we present the structure of the tool suite, and then we show a use case with a simple example.

3.1. The evaluation pipeline

As illustrated in figure 1, we divided the evaluation into 3 stages : Code generation, model annotation and model evaluation.

1. Code generation

   This stage takes as input the native (ie. user-designed) architectural model plus the legacy code, and build the binary code. Code generation relies on OCARINA and POLYORB-HI [14], which is a middleware generator for high-integrity environment. Of course, the same tools must be used for the final binary code generation to preserve computed properties.

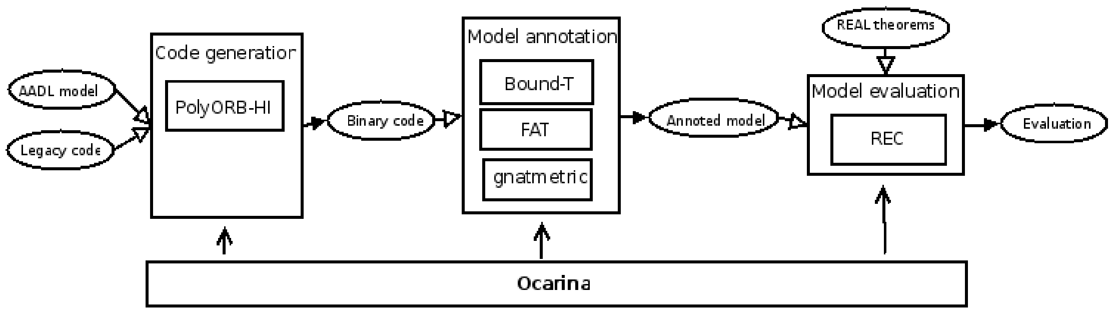


Figure 1. Evaluation tool suite process

2. Model annotation

   This stage takes as input the binary code and the native architectural model, and build an annotated architectural model which takes account of code-specific properties. It relies on tools such as : gnatstack or Bound-T for computing an upper-bound on thread's stack size; Bound-T for extraction of the subprograms' WCET.

   Finally, we use the AADL model manipulation capacities of OCARINA in order to build the annotated model. In case of external tools (Bound-T, gnatstack), glue code has been developed in order to set the tool parameters and extract the results.

3. Model evaluation

   Model evaluation is performed via third-party tools like Cheddar [9] that perform schedulatiblity analysis on a complete AADL model; OSATE [10] for computing latency in the system or performing processor assignment based on thread requirements for CPU usage.

In the following, we illustrate how the toolchain effectively proceeds to evaluate precisely the WCET, and why automation is interesting in that setup.

4. Use case

As an illustration, we selected the `SunSeeker` AADL model, from the OCARINA distribution[2]. `SunSeeker` models a rocket whose goal is to be directed to the sun. It exhibits a traditional subsystem in charge of Guidance, Navigation and Control. This system is made of periodic threads exchanging information on the systems, and commands to be sent to actuators.

This system is amenable to RMA analysis, but requires precise WCET analysis to do so. Besides, even if the overall architecture is already known, the actual computation functions may change as the system evolve. Automating WCET analysis would remove such a time consuming operation, while allowing for early detection of errors when dimensioning the system. In the following, we show how combining stringent code generation strategies and tool can help solving this issue.

4.1. Code generation

Sunseeker relies on AADL semantics for concurrent execution. This model exhibits some periodic execution, on the same processor. Data exchange occurs through Ada protected data component.

[2]See `http://aadl.enst.fr/polyorb-hi/examples.html` for more details

AADL strong semantics allow us to precisely derive supporting runtime entities. OCARINA uses all information in the model to generate minimal code that will support its execution.

Furthermore, in order to be amenable to analysis, the initial model and the code patterns used are both compatible with the Ravenscar profile [1], but also restricts all usage of dynamicity at source-code level (no pointers to procedure, no object orientation, etc.). This ensures the code is deterministic and amenable to analysis. The code to be analyzed encompasses user specific code, execution glue code generate (e.g. threads deduce from application needs), and the supporting kernel and driver. This code, and its compiled counterpart are not enough to allow for WCET analysis.

4.2. Performing WCET analysis

To ensure that WCET analysis can be performed, we must ensure the supporting tools has enough information to proceed. Bound-T inspects object files produced after compilation to evaluate the WCET of a set of function. This analysis relies on a performance model of the processor. We retained the ERC32 processor model, a derivative of the SPARC processor used by ESA. The tool computes all execution path in the system, and return the time to traverse the longest one, if it converges. The latter is the hard point of the analysis.

The code generated by OCARINA strictly follows requirements for high-integrity domain and use the corresponding design patterns for some task artifacts (periodic, sporadic activation patterns, inter-ask communication, . . .). In this setting, Bound-T offers a Hard Real-Time mode that can carefully analyse these patterns. In this context, Bound-T requires the name of the root subprograms on which it must perform WCET analysis: it is the name of the subprogram called at each dispatch time.

Moreover, some of the subprograms used by either legacy code or generated code can be unbounded in term of WCET. To try to analyze them would impede complete WCET analysis. Bound-T allows to specify an *assertion* file that contains the list of subprograms that must not be analyzed, either by providing a well-known WCET (e.g. upper bound for printing a data) or by simply forbidding their analysis (e.g. exception handler). Since generated code follows regular patterns which can be deduced from the AADL model, we can predict which subprograms will ultimately have an unbounded WCET in the binaries. Thus, we generate the assertion file along with the root subprograms from the information found AADL model and the code patterns used by the code generator.

Unbounded WCET occurs when there exist loops in the source code whose upper-bound depends on interactions with other software or hardware elements. We first ensured no portion of generated code has unbounded WCET time per Bound-T analysis. Yet, a typical example is a device driver which completes its work after receiving a signal for the device. We set its value to zero at this level, we tackle this case in a further analysis step.

4.3. Building the annotated model

Once Bound-T completes, we annotate back the AADL model with the information computed by the tool. In hard real-time mode, Bound-T returns an *Execution Skeleton File* (ESF) that stores the result of the analysis. We parse the file to fill the WCET value of each thread. First, we deduce the corresponding AADL thread from the name of its root function's name; then, we compute the exact WCET using the cycles information provided by Bound-T and the the actual processor speed.

Unbounded code cannot be analyzed through Bound-T, we noted it can be either device driver code or user-provided code. The latter is under the responsibility of the designer. The sooner can be accommodated thanks to communication patterns. All task patterns follow a "read-execute-write" cycle. Therefore, we know when a call to the device driver is made. Furthermore, we can deduce from the thread' interface the size of the data to be sent and add it to the WCET of a thread thanks to device metrics information (latency, bandwidth). This ensures one can provide a complete WCET analysis of all the code generated plus user code.

Then, we complete the AADL model with the newly-acquired WCET in the corresponding threads using the `Compute_Execution_Time` property, as seen in code example 1. In this context, we show how Bound-T and Ocarina can be linked in an efficient way. Let us note the process is limited by Bound-T scalability and the complexity of the generated code. Our experiment illustrates it performs correctly in HRT mode thanks to the careful patterns used for inter-task communications.

```
thread implementation Plant_Type.Plant
calls
    plant: subprogram Sunseekerplant_Subprogram.Beacon;
connections
  parameter Controllerinput -> plant.Controllerinput;
  parameter plant.Outputfeedback -> Outputfeedback;
properties
  Dispatch_Protocol  => Periodic;
  Period             => 10 Ms;
  Compute_Execution_Time => 120 Us .. 120 Us; -- Computed by Bound-T
end Plant_Type.Plant;

thread implementation Controller_Type.Controller
calls
  ctrl: subprogram Sunseekercontroller_Subprogram.Impl;
connections
  parameter Outputfeedback -> ctrl.Outputfeedback;
  parameter ctrl.Controllerinput -> Controllerinput;
properties
  Dispatch_Protocol => Periodic;
  Period            => 10 Ms;
  Compute_Execution_Time => 15 Us .. 15 Us;  -- Computed by Bound-T
end Controller_Type.Controller;
```

Listing 1. Sunseeker Threads Implementation

4.4. Completing evaluation

In order to ensure real-time constraints are met, a system schedule must be established. In the context of preemptive tasks, Rate Monotonic Analysis is a quite common solution. Others scheduler such as Earliest Deadline First (EDF), while less common, offer a more optimal usage of the CPU cycles.

An efficient solution in order to verify schedulability is to use Cheddar, a real time scheduling tool which supports popular scheduling methods such as RMA or EDF and takes as inputs AADL files. In order to perform the analysis, Cheddar will need 2 declared properties in the AADL files: `Cheddar_Properties::Fixed_Priority`, which defines the priority assigned to the task; `Compute_Execution_Time`, which defines the WCET of the thread root subprogram, as computed by Bound-T and the analysis of the communication pattern. To complete some analysis, one may also to define other time-impacting non-functional properties such as `Period`, `Deadline` or `Dispatch_Protocol`. Now that the thread root program WCET has been computed in the annotation phase, Cheddar can perform scheduling analysis and conclude to system schedulability.

Another solution in order to analyze the scheduling is to perform a Response Time Analysis (RTA). While RMA indicates whether the tasks can meet their deadline or not, RTA gives a value for the total task WCET, including perturbating ones such as context switch due to preemption, blocking, clock signal, etc. [4]. The worst-case response time of a process is defined by the time it takes for that process to complete its most demanding set of activities in response to a single activation event occurring under maximum contention from the rest of the system. The worst-case response time of a given activity (ie. a task) can be computed by a recurrence on the sum of three components, which are: the worst-case execution time of the task; the interference incurred by the task; the blocking experienced by the task.

Authors in [12] give a complete definition of each term involved in Response Time Analysis (RTA), and provides formal expressions in order to compute them within a Ravenscar-compliant model. An important thing about these expressions is that they involve the knowledge of the subprogram WCET and the worst-case time of some framework primitives or actions such as a context switch or an clock interrupt handler duration. The run-time environment we used is based on the same mechanisms as the one developed in this analysis. We can then reuse these equations and validate the system. In this context, knowing precisely when the different runtime primitives are used is relevant. We have developed REAL[3], a domain-specific language that allows one to perform queries and computations on AADL models. We implemented a set of REAL subprograms which allows to proceed to full RTA by taking into account the different impacting factors of an RTA analysis. Similar extensions can be written, to take into account other platform-specific or domain-specific analysis.

4.5. Assessment and Related work

The complete process proved to be efficient to analyse directly model. This is mostly due to the fact that Bound-T and the Ocarina code generator relies on the same family of patterns. Writing the annotations for Bound-T is therefor natural.

The complete automation leads to a shortened development time: developer needs only to focus on a high-level design representing its system, joint work by Ocarina, Cheddar and Bound-T allows the designer to validate its system directly. Yet, the validation is done atomically for a complete system. Incremental validation on subparts is a current work in progress.

The combination of a modeling toolsuite and WCET analysis tools is an interesting feature for the system designer: he can test various configurations automatically. This has been already tested in different settings.

Authors in [5] discuss the integration of a WCET tool in Matlab/Simulink. The approach developed is notionally equivalent. The key difference resides in the code to be analyzed. In the case of Matlab, the code is highly sequential whereas our code generator introduces Ada concurrent entities (threads and protected objects). In both cases annotations are built from an a priori knowledge of the code generated. Our approach exploits a higher level description language, AADL, that can integrate multiple languages to implement its functional blocks (e.g. SCADE, C or Ada). We extend this work to a more complete engineering framework.

[3]REAL sources and documentation can be accessed from http://aadl.enst.fr/ocarina/real.html

5. Conclusion

In this paper, we discussed the issue of performing WCET analysis for complex systems. We noted that such analysis is required, yet it is time consuming. We linked this activity to schedulability analysis, usually performed on a high-level model of a system.

In this context, we proposed to use the same model to 1/ generate code, and 2/ perform WCET analysis on the code generated to 3/ refine the model with precise WCET values for the system. We proposed a complete process based on the AADL modeling notation and the AADL toolsuite we developed. AADL defines precise semantics for all its constructs, this allows one to derive precise code and provide roots for analysis.

Typical WCET analysis tools require “hints” to point the code executed by the processing threads, the potential infinite loops of these threads, path irrelevant to the analysis, etc. By computing these information as part of the code generated, and then passing it to the Bound-T analysis tool, we shorten the distance between model and executable system, allowing for precise analysis as the model evolves.

This provides one step forward a complete toolchain to build critical real-time systems from high-level models, combining precise descriptions of the system resources, code generation with high-integrity restrictions enforced, precise WCET analysis, and the capability of performing schedulability analysis or performance evaluation in a uniform framework.

Future work will consider the extension of this toolsuite to exploit all computed information to perform model-based optimizations of the system, e.g. computing precisely the minimal number of threads to provide an semantically-equivalent system, reducing latency, etc.

References

[1] B. Dobbing, A. Burns, and T. Vardanega. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Technical report, 2003.

[2] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, CMU, 2006. CMU/SEI-2006-TN-011.

[3] J. Hugues, B. Zalila, and L. Pautet. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP’07)*, pages 106–112, Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.

[4] M. Josef and P. Pandya. Finding response times in real-time systems. *BCS Computer Journal 29(5)*, pages 390–395, 1986.

[5] R. Kirner and P. Puschner. Integrating WCET analysis into a matlab /simulink simulation model. In *Proceedings ot the 16th IFAC Workshop on Distributed Computer Control Systems, Sydney, Australia, School of Computer Science and Engineering, UNSW.*, November 2000.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in hard-real-time environment. In *Journal of the ACM*, january 1973.

[7] Tidorum Ltd. Bound-T Execution Time Analyzer, url: `http://www.bound-t.com`.

[8] SAE. Architecture Analysis & Design Language(AS5506). sep 2004.

[9] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, New York, USA, December 2004. ACM Press.

[10] SEI AADL team. Osate : an extensible source aadl tool environment. Technical report, SEI, December 2004.

[11] S. Ureña, J. Pulido, J. Zamorano, and J. A. de la Puente. Handling Temporal Faults in Ada 2005. In Springer Verlag, editor, *Proceedings of the 12th Reliable Software Technologies AdaEurope'07*, 2007.

[12] T. Vardanega, J. Zamorano, and J. A. de la Puente. On the dynamic semantics and the timing behavior of ravenscar kernels. *Real-Time Syst.*, 29:59–89, 2005.

[13] T. Vergnaud and B. Zalila. Ocarina: a Compiler for the AADL, `http://aadl.enst.fr`. Technical report, Télécom Paris.

[14] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Proceedings of the 11th IEEE International ymposium on Object-oriented Real-time distributed Computing (ISORC'08)*, Orlando, Florida, USA, May 2008. IEEE Computer Society Press.

COMPUTING TIME AS A PROGRAM VARIABLE: A WAY AROUND INFEASIBLE PATHS

Niklas Holsti[1]

Abstract

Conditional branches connect the values of program variables with the execution paths and thus with the execution times, including the worst-case execution time (WCET). Flow analysis aims to discover this connection and represent it as loop bounds and other path constraints. Usually, a specific analysis of the dependencies between branch conditions and assignments to variables creates some representation of the feasible paths, for example as IPET execution-count constraints, from which a WCET bound is calculated. This paper explores another approach that uses a more direct connection between variable values and execution time. The execution time is modeled as a program variable. An analysis of the dependencies between variables, including the execution-time variable, gives a WCET bound that excludes many infeasible paths. Examples show that the approach often works, in principle. It remains to be seen if it is scalable to real programs.

1. Introduction

Static WCET analysis is usually divided into three main parts [16]: *Flow analysis* models the possible execution paths (instruction sequences) as a control-flow graph (CFG). *Processor-behaviour analysis* bounds the execution time of each basic block in the CFG. *Bounds calculation* finds bounds on the execution time of entire execution paths. WCET tools often also make some analysis of the possible values of program variables. This *value analysis* supports the data-dependent parts of flow analysis and processor-behaviour analysis.

Define the *structural paths* as all paths through the CFG assuming that any conditional branch can be taken or not taken – as though the conditions could have any value, at any time. Most programs have loops and thus an infinite number of structural paths. Even after loop-bounds are applied to make a finite set of paths, often many logically infeasible paths remain, leading to an overestimated WCET bound. An *infeasible path* here means a connected sequence of CFG elements – nodes and edges – that contains an edge for which the condition must evaluate to *false* if the path is executed up to this edge and all earlier edge conditions on the path are *true*.

Flow analysis can detect some infeasible paths, *eg*. [2, 14]. Typically, the computations (expressions, assignments, branch conditions) are analysed and correlated to create some representation of the feasible and infeasible paths (*eg*. dead code or mutually exclusive basic blocks). This “flow fact” representation is then fed into the bounds calculation, for example as execution-count constraints for IPET ([6] and other references in [16]). The connection between the computation and the execution time is thus indirect: first from the computation to the flow facts, and then from the flow facts to the execution time. This paper explores another approach with a more direct connection. Section 2

1 Tidorum Ltd, Tiirasaarentie 32, FI 00200 Helsinki, Finland

discusses the dependencies between values computed in a program, and how these lead to infeasible paths. Section 3 presents the idea of the paper: to model the execution time as a program variable, subject to dependency-sensitive value analysis. Section 3 shows analysis results for some examples of infeasible paths, using a value analysis based on Presburger sets [12]. Section 4 describes this analysis for loop-less code. Section 5 extends the Presburger-set analysis to loops. Section 6 considers how the loop analysis handles the execution-time variable and infeasible paths involving loops. Section 7 closes the paper with a review of related work and a discussion.

2. Dependencies Between Variables and Values

Most variables in a program get their values from expressions that use the values of other variables, which leads to dependencies between the values of these variables. Dependencies also arise when different variables are independently computed and assigned within the same control structure. For example, in an **if-then-else** structure, the values assigned in the **then** branch occur together, and together with the *true* value of the condition, while those in the **else** branch occur together with the *false* value, but mixtures are infeasible.

Several static program analyses discover such dependencies; *eg*. [1]. Few WCET tools do it, but Lisper has suggested it [8]. Dependencies are the very reason for infeasible paths. Dependencies between loop induction variables and the looping condition lead to loop repetition bounds. The classic example of an infeasible path is two consecutive **if-then-else** structures with inter-dependent conditions.

The Bound-T WCET tool [5] models variable values as sets of integer tuples constrained by Presburger formulae – *Presburger sets* for short. Each element in a tuple models one variable; a tuple models one possible combination of variable values; and a set of tuples models all possible combinations of variable values. Dependencies between variables are included because the Presburger formulae constrain the whole tuple, not each element (variable) separately. For example, for two variables x, y, the set $\{[x, y] \mid x > 5 \text{ and } y < 3x\}$ models a state where x has any value greater than 5 and y has any value less than 3x. Operations on Presburger sets include set intersection, set union, set complement, relational join (mapping, application), convex hull, and test for subset [12]. Bound-T currently uses its Presburger model mainly for loop-bound analysis and does not search for other kinds of infeasible paths. This paper suggests how a dependency-sensitive value-analysis, such as the one in Bound-T, can be used to compute WCET bounds that exclude infeasible paths, without explicitly finding and representing the infeasible or feasible paths themselves. In fact, a separate bounds-calculation phase is not needed; value analysis produces WCET bounds.

3. Execution Time as a Computed Value

If we can analyse dependencies between variables, and we want to analyse the dependency between variables and execution time, perhaps we can get there by treating the execution time as a variable in the computation. To do so, augment the program (for the analysis only) with a new global variable, T say, that represents the execution time from the start of the program (or the subprogram, for a modular analysis). Augment each basic block b with the assignment $T := T + t(b)$, where $t(b)$ is the execution time of the basic block, as found by processor-behaviour analysis using the structural paths. If the basic block can have a range of execution times, use interval arithmetic. Assume that `T` is initially zero. Use a value analysis to find the values of `T` at the end of the program; these are the execution times of the program. If the analysis models dependencies between variables, the final bounds on `T` should exclude some or all infeasible computations, depending on the precision of the analysis of the (indirect) dependencies between `T` and branch conditions.

3.1 Example: Condition after Condition

The classic example of infeasible paths is the correlated pair of **if-then-else** conditional statements:

```
if x < 1 then <compute for 100 cycles>;
         else <compute for 10 cycles>; end if;
<code that does not change x, for 30 cycles>;
if x > 3 then <compute for 200 cycles>;
         else <compute for 20 cycles>; end if;
```

(For clarity, the time for evaluating the conditions is included in the times for the **then** and **else** branches.) Each conditional statement in this example has a fast branch and a slow branch. The structural paths include the path that takes both the slow branches, totalling 100 + 30 + 200 = 330 cycles. However, this path is logically infeasible because the conditions for the two slow branches, `x` < 1 and `x` > 3, cannot be true at the same time. The longest feasible path occurs when `x` > 3 and gives 10 + 30 + 200 = 240 cycles. Now augment the code with the execution-time variable `T` :

```
T := 0;
if x < 1 then <compute for 100 cycles>;  T := T + 100;
         else <compute for 10 cycles>;   T := T + 10; end if;
<code that does not change x, for 30 cycles>;  T := T + 30;
if x > 3 then <compute for 200 cycles>;  T := T + 200;
         else <compute for 20 cycles>;   T := T + 20; end if;
```

A value analysis of the augmented program with the Presburger method (to be described in Section 4) gives a model of the final values of the variable tuple [`x`, `T`] as the three-part set:

$$\{[x, T] \mid x < 1 \text{ and } T = 150\} \cup \{[x, T] \mid 1 \le x \le 3 \text{ and } T = 60\} \cup \{[x, T] \mid x > 3 \text{ and } T = 240\}$$

The analysis thus excludes the infeasible value `T` = 330 and gives the precise bounds `T` = 60 .. 240.

3.2 Example: Saturating a Value

This example code makes sure that the variable `x` does not exceed the interval `min` .. `max`:

```
if x < min then x := min; end if;
if x > max then x := max; end if;
```

This code has an infeasible path if `min` ≤ `max`, because the assignment `x` := `min` makes the condition in the second **if** false. (The infeasible path could be avoided with an **else if**, but some programmers seem to prefer the above form.) Augment the program with an execution-time variable `T` as follows, adding **else** branches just to model the condition-evaluation time:

```
T := 0;
if x < min then x := min; T := T + 3; else T := T + 1; end if;
if x > max then x := max; T := T + 3; else T := T + 1; end if;
```

Assume for simplicity that `min` = 1 and `max` = 10. The model of the final values of [`x`, `T`] is then:

$$\{[x, 2] \mid 1 \le x \le 10\} \cup \{[1, 4]\} \cup \{[10, 4]\}$$

Thus, the infeasible value `T` = 6 is avoided, and the precise range `T` = 2 .. 4 is computed. The correct range for `T` is computed even if the actual values of `min` and `max` are unknown (variable), as long as the analyser knows that `min` ≤ `max`. The path-exclusion analysis of Stein and Martin [14] cannot handle this example because the assignment `x` := `min` does not dominate the second conditional. However, as Stein and Martin say, their analysis could be extended to such "non-linear slices" at the cost of more complex Presburger problems and increased analysis time.

4. Presburger Analysis Basics

This section describes the Presburger-set value-analysis that gives the above results, taking the saturation code in section 3.2 as an example. The analysis is similar to the one in Bound-T [5].

In the absence of loops the analysis is quite simple. Assume that there are *n* variables. At the start of the program the model is the universal set of all possible variable-value tuples Z^n, where *Z* is the set of all integers. Our example has the variables `x` and `T` so *n* = 2.

Each statement (in Bound-T: each machine instruction) is modeled as a transfer relation between the *n* input values and the *n* output values, a Presburger subset of $Z^{2n} = Z^n \times Z^n$. For example, the statement `T` := 0 is modeled by the transfer relation {[`x`, `T`, `x'`, `T'`] | `x'` = `x` and `T'` = 0}. Here the symbols `x` and `T` represent the variable values before the statement, while the primed symbols `x'` and `T'` represent the values after the statement. This relation can be written more compactly as {[`x`, `T`] → [`x`, `0`]} where the arrow separates the "before" and "after" values and the constraints are implied by the expressions. The universal 2-tuple set Z^2 is then written {[`x`, `T`]}.

Starting from the universal set {[`x`, `T`]} and applying the transfer relation {[`x`, `T`] → [`x`, `0`]} produces the set of new variable values {[`x`, `0`]}, which models the variable values after the statement `T` := 0. In this way, the variable-value model is propagated over the statements and nodes in the CFG. To propagate the model over a CFG edge, the model set is intersected with the set defined by the edge condition. When several edges converge to the same node, the union of the sets from each incoming edge is computed. The union set is often non-convex, which increases both the power and the complexity of the analysis, and differs from the model in [1]. Condition flags are another source of non-convex sets, because their values are constrained by disjunctive formulas.

Figure 1 illustrates this analysis for the example in section 3.2. The figure shows the CFG with Presburger sets placed before and after nodes to show the value-model at these points. Note the edge from the second **if** towards the right, when `x` > 10; this edge condition eliminates the tuple [1, 3] from the value set, which reflects the mutual exclusion of the two CFG nodes on the right. If the analysis gives an empty Presburger set at some point – which does not happen in this example – that part of the CFG is infeasible (dead).

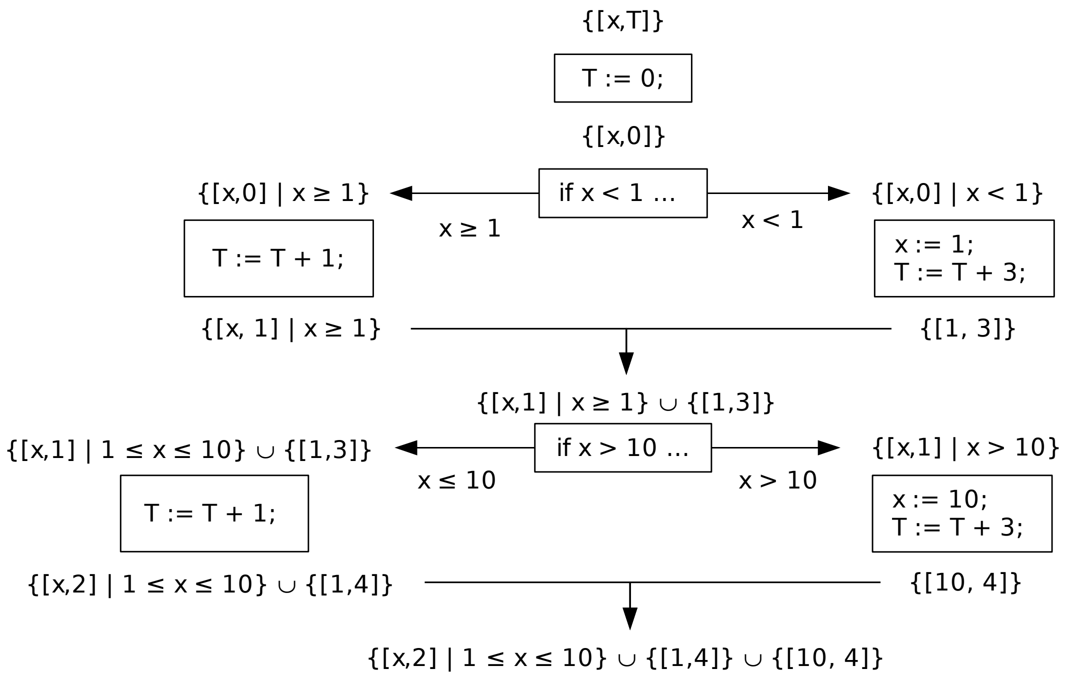


Figure 1: Presburger analysis of the saturation example

Computation with Presburger sets has exponential complexity in general. For practical purposes it is important to minimize the number of constraints and union operations. If the `T` variable is updated separately in each branch of a conditional, as above, `T` depends on every branch condition. For a conditional where the branches have similar execution times, one could instead update `T` once for the whole conditional, with `T := T +` *max* (**then** branch, **else** branch), as in the *timing schema* methods [13]. The complexity can thus be reduced by a selective fall-back to a timing schema.

5. Presburger Analysis of Loops

This section extends the Presburger-set analysis to loops, showing how it can find bounds on the number of iterations and bounds on the variables after the loop. As a running example, consider this code that reverses the order of elements `n` .. `n` + 9 of the vector `vec`, where `n` is a variable:

```
i := n; j := n + 9;
while i < j loop
   z := vec[i]; vec[i] := vec[j]; vec[j] := z;
   i := i + 1; j := j – 1;
end loop;
```

The analysis will ignore the values in `vec` because they have no effect on the loop or its execution time, and pointer analysis is out of scope for this paper. The modeled variables are `i`, `j`, `n`, and `z`. Since the `vec` values are ignored, the assignment to `z` is modeled as storing an unknown value in `z`.

Assume, initially, that there is only one loop, with a single *loop head node* that is the single point of entry to the loop. The analysis is based on the *repetition relation* of the loop. One pass through the loop body from the loop head up to and including a back edge is called a *repetition* of the loop. The repetition relation is the Presburger relation that shows how one repetition changes variable values. The repetition relation of the example loop above is {[i, j, n, z] → [i+1, j-1, n, z'] | i < j}, where z', the new value of z, is unconstrained. Similarly, a pass through the loop body from the loop head to an edge that leaves the loop is called an *exit* from the loop. The full execution of a loop is a sequence of zero or more repetitions, followed by one exit.

5.1 Classifying Variables as Invariant, Induction, or Fuzzy

For a program with loops, static analysis needs some form of induction or least-fixpoint iteration. The Presburger-set domain has infinite ascending chains, so least-fixpoint iteration cannot be used directly. Instead, we analyse the repetition relation of the loop to divide the variables into three classes: an *invariant* variable is not changed at all; an *induction* variable is changed by a bounded *increment* (possibly negative); and the rest are *fuzzy* variables that change in other ways.

Let R be the repetition relation of a loop and x a variable. Introduce a new variable dx on the domain side to represent the possible increment in x, making R a relation in $Z^{n+1} \times Z^n$. This role of dx is expressed by the relation Rd = R ∩ {[..., x, ..., dx] → [..., x + dx, ...]}. Compute bounds on dx by taking the domain of Rd, then projecting away the other variables to leave only the set of dx values, and finally computing the convex hull of this one-dimensional set. The convex hull is the interval that bounds dx. If dx is bounded to zero, x is invariant; otherwise, if dx is bounded to a finite interval, x is an induction variable; otherwise x is a fuzzy variable. Note that dx can depend on the other variables, but the projection and convex-hull operations discard those dependencies in order to arrive at *constant* lower and upper bounds on dx. This is a necessary approximation at this step of the analysis because the next step uses the bounds on dx to multiply a variable, and the Presburger model allows multiplication only when at most one of the factors is a variable.

It is practical to introduce all increment variables at once. In the vector-reversal example, this introduces the variables di, dj, dn, and dz and gives the extended loop repetition relation

```
Rd = {[i, j, n, z, di, dj, dn, dz] → [i+1, j-1, n, z'] | i < j}
   ∩ {[i, j, n, z, di, dj, dn, dz] → [i+di, j+dj, n+dn, z+dz]}
```

Computing bounds on di, dj, dn, dz from this Rd shows that di = 1, dj = -1, dn = 0, and dz is not bounded. Thus i and j are induction variables, n is invariant, and z is a fuzzy variable.

5.2 Bounding the Number of Loop Repetitions

Introduce a new variable c to model the iteration number 0, 1, 2, ... The value-model at the start of the loop-head node is a Presburger set where each invariant variable has its initial (and invariant) value, each induction variable equals its initial value plus c times its increment, and all fuzzy variables are unconstrained[2]. Propagate the model from the loop head to all parts of the loop as in section 4. Next, check if the Presburger sets on the back edges or exit edges of the loop (which include the repetition or termination conditions) imply bounds on c; if so, these are the loop repetition bounds.

2 A better but more complex model includes variable dependencies from the preceding iteration. Bound-T does so.

For the vector-reversal example, the Presburger set on the loop back edge is

$$\{[i, j, n, z, c] \mid i-1 < j+1 \text{ and } i = n+1+c \text{ and } j = n+8-c\}$$

Here `i` and `j` represent the variable values at the end of the loop body, after they have been incremented by `di` and `dj`. Projecting this set to the variable `c` and computing the convex hull gives the constraint $c \leq 4$, which shows that the loop is repeated at most 5 times (for $c = 0 .. 4$). Of course, if the number of repetitions of a loop is bounded by a user annotation, it is unnecessary to compute bounds on `c`, as they are given by the annotation.

5.3 Modeling Variables After the Loop

The final step of the Presburger analysis of a loop is to model the variable values after the loop as the result of some number of loop repetitions followed by a loop exit. For an induction variable, the repetitions are modeled by multiplying the number of repetitions with the increment of the variable. Both factors have constant bounds, but simply multiplying these constants would hide possible dependencies between the increments, the number of loop repetitions, and other variables. At this point there is a choice: one, but not both, of the factors can be treated as a Presburger variable, making the analysis sensitive to dependencies between this variable and other variables. I have not found a way to compute the number of repetitions as a Presburger variable, only as a constant extracted from the Presburger model (bounds on the iteration number `c`), and therefore I choose the increment factor as the Presburger variable, leaving the number of loop repetitions as a constant factor. In the subsequent analysis of the post-loop code, the auxiliary variables for the increments can be projected away; their relationships, if any, are still encoded in the structure of the projected set.

In the vector-reversal example, there are no dependencies between the increments and other variables. Similar analysis (omitted here) shows that the minimum number of repetitions is also 5, so the values after the loop are modeled by the set $\{[i, j, n, z] \mid i = n+5 \text{ and } j = n+4\}$.

For a loop that has more than one entry point, in other words more than one head node, a repetition is defined as a pass through the loop from some head node to some back edge, and an exit is defined as a pass from some head node to some edge that leaves the loop. The repetition relation is defined as the union of the transfer relations from all possible repetition paths, and the set of initial values as the union of the initial-value sets at all head nodes. The analysis itself remains the same.

If there are nested loops the Presburger-set analysis has two phases. The first phase analyses the loops bottom-up, from the innermost to the outermost, classifies the variables as invariant, induction, or fuzzy for each loop, and makes an approximate model of the total effect of each loop as a relation that uses this classification but an unknown number of loop repetitions. The repetition relation of an outer loop uses these approximations for all inner loops. The second phase analyses the loops top-down, from outermost to innermost, and in flow order for loops at the same level. This second phase computes loop-repetition bounds and better post-loop value models that include bounds on the induction-variable increments and bounds on the number of repetitions. In some cases the models can be further improved by iterating these two phases a few times. Note also that user annotations that place bounds on variable values, at one program point or everywhere, are quite easy to include in the Presburger sets, apart from the practical problem of mapping source-level variable identifiers to machine-level storage locations.

6. Execution Time of Loops

This section considers how the Presburger-set analysis of loops applies to the execution-time variable `T`, and in particular whether the bounds on `T` include or exclude infeasible paths in loops. The execution-time variable `T` is evidently an induction variable, and its increment `dT` is the execution time of the loop body. From section 5 the model of `T` after the loop is the initial value T_0 of `T` (the cumulated execution time up to the start of the loop) plus `dT` times the number of loop repetitions, plus the execution time of loop exit. This is roughly the same formula as in the timing schema methods [13] but here both T_0 and `dT` can depend on other variables. The exit from a loop is analysed using the methods in section 4 and is already “outside” the loop from the viewpoint of loop analysis. Thus, only loop repetitions are discussed here.

Loops can involve many kinds of dependencies and infeasible paths. For some kinds, the proposed analysis excludes infeasible paths; for others it does not, as discussed below.

6.1 Iteration-Independent Dependencies

A dependency or an infeasible path is *iteration-independent* if it applies on every repetition of the loop. In the simplest case, the loop contains a path that is infeasible on every repetition, whatever happens outside the loop or in other repetitions or in an exit from the loop. An example is a loop that contains the “saturation” code from section 3.2. The Presburger analysis omits the infeasible path from `dT`, and so also from the final value of `T`.

Another example is a conditional outside the loop that conflicts with a conditional inside the loop, on every repetition of the loop. For example, put a loop around the second **if-then-else** statement in the example in section 3.1, keeping the variable `x` as a loop invariant, thus:

```
t := 0;
if x < 1  then t := t + 100;
          else t := t +  10; end if;
for i in 1 .. 7 loop
   if x > 3  then t := t + 200;
             else t := t +  20; end if;
end loop;
```

In the repetition relation Rd the `T`-increment `dT` now depends on `x`, as can be seen in the domain of Rd projected to the variables [`x`, `dt`], which is the set $\{[x, 20] \mid x \le 3\} \cup \{[x, 200] \mid x > 3\}$. If execution takes the slow branch of the first conditional, before the loop, it must then take the fast branch of the second conditional, inside the loop, on every loop iteration. This dependency is reflected in the model of the values after the loop. Projected to the variables [`x`, `T`] this set is

$$\{[x, 240] \mid x < 1\} \cup \{[x, 150] \mid 1 \le x \le 3\} \cup \{[x, 1410] \mid 4 < x\}$$

The infeasible path where both slow branches are taken would give a final `T` of $100 + 7 \times 200 = 1500$ cycles, which this analysis evidently excludes. The worst case takes the fast branch before the loop and the slow branch in every repetition of the loop, giving $10 + 7 \times 200 = 1410$ cycles.

6.2 Iteration-Specific Dependencies

Some dependencies and infeasible paths apply only on some repetitions. For example, put a loop around the second **if-then-else** statement in the example in section 3.1, but also add a statement in the loop that changes `x` on repetition 4. Now the combination of the two slow branches is infeasible for repetitions 0 through 3, but may be feasible for later repetitions. The Presburger analysis cannot exclude these infeasible paths because the bounds on `dT` are computed over all repetitions together.

When a loop contains important iteration-dependent "unbalanced" conditionals, it is possible to introduce new variables to model how often each branch is executed, and compute the final `T` as the sum of these execution frequencies times the execution times (`dT`'s) of the branches. This is very similar to the IPET formulation, but now the `dT`'s of the branches can depend on other variables, so iteration-independent infeasible paths can be excluded. The execution frequencies could perhaps be computed analytically from the Presburger model [11] or by other methods [4, 8].

6.3 Dependent Loop Bounds

Important infeasible paths may arise when loop bounds correlate with conditionals, as here:

```
if <cond> then <compute for 100 cycles>; n :=  5;
          else <compute for  10 cycles>; n := 20; end if;
for i in 1 .. n loop <compute for 10 cycles>; end loop;
```

A normal WCET analysis includes the infeasible path where the slow **then** branch is followed by 20 iterations of the loop, for a total of 300 cycles, although this branch implies only 5 iterations and 150 cycles. The actual WCET is 210 cycles, for the fast **else** branch followed by 20 iterations. The analysis of this paper does no better and reports a WCET bound of 300 cycles, because the bound on the number of loop repetitions is modeled as a constant (20), not as a Presburger variable (`n`) that can depend on other variables. As explained in section 5.3 the loop bound and the increment `dT` cannot *both* be Presburger variables, because the Presburger model does not allow multiplication of two variables. Thus, one cannot at the same time model the dependencies of `dT` and of the loop bound.

7. Discussion

The two main points in this paper are to model execution time as a program variable, and to use a dependency-sensitive value-analysis to exclude infeasible values. Neither point is novel in itself, but the combination is new, as far as I know. Haase [3] defined execution time as a program variable, for proving real-time properties using weakest-precondition calculus. Haase's work influenced Shaw's definition of timing schemata [13], but the *max* (**then**, **else**) schema for conditionals disconnects execution time from other variables. The annotation language described by Mok *et al.* [9] is a kind of program to compute the execution time, but it is evaluated apart from the target program.

Model-checking approaches to timing analysis [10] must treat execution time as a variable, and may use semi-symbolic state representations. The analysis of Cousot and Halbwachs [1] identifies affine dependencies between variables, but uses convex hulls, which I believe means that its ability to exclude infeasible paths is weak. Lisper included dependency-sensitive value analysis in his ambitious proposals [8], but without execution time as a variable. Stein and Martin use Presburger models in their analysis of exclusive paths [14]. However, they create constraints for bounds calculation by IPET, and do not use a time variable.

The method explored in this paper needs a name. Let us call it the *time-variable* method. Different value analyses can be used with this method; the Presburger analysis is only one possibility. The examples show that the time-variable method with Presburger analysis excludes some kinds of infeasible paths, but not all kinds. The method is in principle applicable to more kinds of of infeasible paths than some other methods, such as [14]. The time-variable method does not need a representation of feasible and infeasible paths (a flow-fact language) nor a separate bounds-calculation phase. A drawback is that it does not, by itself, exclude infeasible paths from the processor-behaviour analysis. This could lead to over-estimated WCETs of CFG elements. However, while it is easy to exclude single infeasible (dead) nodes and edges from the processor-behaviour analysis, it seems much harder to exclude longer infeasible paths, because the analysis would have to identify which paths generate which processor states. Hence this drawback may be insignificant.

In processors with caches and other accelerators the execution time $t(b)$ of a basic block b is not constant but depends on the history of execution. Using a history-independent worst-case $t(b)$ can be a gross over-estimation. Current WCET tools have two approaches to this problem. The first approach (virtually) expands the CFG so that the same basic block is represented by different CFG nodes in different contexts, and thus by different context-specific values of $t(b)$ [15]. The most common expansion peels the first iteration from each loop so as to separate the I-cache misses that may occur on the first iteration, from the I-cache hits that are assured on later iterations. The time-variable method allows such CFG expansions and their benefits are the same as in the IPET method.

The second approach to handle history-sensitive accelerators is to expand the IPET problem by new ILP variables that model the states of the accelerator mechanisms (*eg*. the contents of the I-cache) and the additional execution time required by accelerator state transitions (*eg*. an I-cache miss) [7]. As the time-variable method does not use IPET, it cannot use this approach as such. In principle, new Presburger variables could model accelerator states, but the complexity would likely be impractical.

Experience with the Bound-T tool [5] shows that the Presburger analysis is practical in many cases for analysis of loop bounds, but also impractical in some cases (large subprograms), at least in its current implementation. I think that the complexity comes mainly from the disjunctions (set unions). In [14] Stein and and Martin restrict their analysis to "linear" slices and thus avoid disjunctions. Still, I believe that disjunctions are necessary for analysing several kinds of infeasible paths. Better slicing to simplify the Presburger model, selective use of a timing schema for well-balanced conditionals, and perhaps taking the convex hull of the Presburger sets at selected program points may reduce the analysis time. Bound-T currently calculates WCET bounds with IPET, not with the time-variable method. Implementation and experimental evaluation of the time-variable method is future work.

8. References

[1] COUSOT, P., HALBWACHS, N., Automatic discovery of linear restraints among variables of a program, in: Proc. of the 5th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, January 1978.

[2] GUSTAFSSON, J., ERMEDAHL, A., SANDBERG, C., and LISPER, B., Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution, in: Proc. of the 27th IEEE International Real-Time Systems Symposium (RTSS'06), December 2006.

[3] HAASE, V.H., Real-time behaviour of programs, in: IEEE Transactions on Software Engineering, Vol SE-7, No. 5, September 1981.

[4] HEALY, C., SJÖDIN, M., RUSTAGI, V., WHALLEY, D., and ENGELEN, R. V., Supporting Timing Analysis by Automatic Bounding of Loop Iterations, in: Real-Time Systems, vol. 18, no. 2-3, May 2000.

[5] HOLSTI, N., Bound-T Reference Manual. Tidorum Ltd, http://www.bound-t.com/, February 2008.

[6] LI, Y.-T. S., MALIK, S., Performance analysis of embedded software using implicit path enumeration, in: Proc. of the 32:nd Design Automation Conference. 456–461, 1995.

[7] LI, Y.-T. S., MALIK, S. WOLFE, A., Cache modeling for real-time software: beyond direct mapped instruction caches, in: Proc. of the 17th IEEE Real-Time Systems Symposium, December 1996.

[8] LISPER, B., Fully automatic, parametric worst-case execution-time analysis, in: Proc. of the 3rd International Workshop on Worst-Case Execution Time Analysis, Porto, Portugal, July 2003.

[9] MOK, A.K., AMERASINGHE, P., CHEN, M., and TANTISIRIVAT, K., Evaluating tight execution time bounds of programs by annotations, in: Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software, Pittsburgh, PA, USA, May 1989.

[10] NAYDICH, D., GUASPARI, D., Timing analysis by model checking, in: Lfm2000 – Fifth NASA Langley Formal Methods Workshop, Technical Report NASA-2000-cp210100.

[11] PUGH, W., Counting solutions to Presburger formulas: how and why, in: Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, Orlando, Florida, United States, June 1994.

[12] PUGH, W., et al., The Omega project: frameworks and algorithms for the analysis and transformation of scientific programs, University of Maryland, http://www.cs.umd.edu/projects/omega.

[13] SHAW, A.C., Reasoning about time in higher-level language software, in: IEEE Transactions on Software Engineering, Vol 15, No 7, July 1989.

[14] STEIN, I., MARTIN, F., Analysis of path exclusion at the machine code level, in: Proc. of the 7th International Workshop on Worst-Case Execution Time Analysis, Pisa, Italy, July 2007.

[15] THEILING, H., FERDINAND, C., Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis, in: Proc. of the 19th IEEE Real-Time Systems Symposium, December 1998.

[16] WILHELM, R., et al., The worst-case execution time problem — overview of methods and survey of tools, in: ACM Transactions on Embedded Computing Systems, Volume 7, Issue 3, April 2008.

IMPROVING THE WCET COMPUTATION TIME BY IPET USING CONTROL FLOW GRAPH PARTITIONING

C. Ballabriga, H. Cassé[1]

Abstract
Implicit Path Enumeration Technique (IPET) is currently largely used to compute Worst Case Execution Time (WCET) by modeling control flow and architecture using integer linear programming (ILP). As precise architecture effects requires a lot of constraints, the super-linear complexity of the ILP solver makes computation times bigger and bigger. In this paper, we propose to split the control flow of the program into smaller parts where a local WCET can be computed faster - as the resulting ILP system is smaller - and to combine these local results to get the overall WCET without loss of precision. The experimentation in our tool OTAWA with lp_solve solver has shown an average computation improvement of 6.5 times.

1. Introduction

Hard real-time systems are composed of tasks which must imperatively finish before their deadlines. To guarantee this termination, scheduling analysis requires the knowledge of each task WCET. To obtain this WCET, three steps are necessary: (1) the task control flow analysis, which determines the possible program paths, (2) the architecture effects analysis, which takes into account the various hardware components (CPU pipeline, instruction cache, etc) to produce timings for program paths, and (3) the final WCET computation.

A widely-used approach for the last step is the Implicit Path Enumeration Technique (IPET) [10]. The different components of the computation (control flow, pipeline execution, etc) are represented as integer linear constraints and the WCET as a linear expression to maximize. The result is then obtained using an Integer Linear Programming (ILP) solver that consumes a lot of time, and is often one of the most time consuming tasks in the WCET computation. The resolution time is usually a function (often non linear) of the number of constraints and variables in the ILP system. For example, the approach for instruction caches of Ferdinand [7] provides good results but the virtual loop unrolling multiplies the number of variables and constraints of a block nested in n level of loops by 2^n.

In this paper, we attempt to reduce the time cost of the WCET computation by splitting the WCET computation into smaller ILP systems and by merging the intermediate results. According to the built constraints, the program is split into small regions where a local safe WCET can be computed. This result is then re-used to compute bigger regions until the whole program is covered. Notice that the first two steps - control flow and architecture effects analyses - are left untouched. This approach has shown time computation improvements in different contexts (taking into account various architecture effects) for the ILP solver lp_solve.

[1]IRIT, Université de Toulouse, CNRS - UPS - INP - UT1 - UTM, 118 rte de Narbonne, 31062 Toulouse cedex 9, email: {ballabri,casse}@irit.fr

The approach is related to [5], where the author partitions the WCET computation into *fact clusters*. The fact clusters are composed of sets of flow facts and the scopes they are applied to (a scope is a structured CFG subgraphs). The clusters bound the range of flow facts in order to get an accurate and feasible WCET computation. Our approach uses a different method to partition the CFG (producing smaller regions) and to take hardware effects into account (especially the pipeline and the instruction cache) without loss of precision.

In the next section, we describe precisely the ILP constraints generated to model the control flow of the task and the architecture effects. Then we describe our approach for splitting the WCET computation and, in particular, we describe the problems caused by hardware modeling constraints for local effects and global effects. The CPU pipeline and the instruction cache are taken as examples. The next section presents the experimentation results of our method and we conclude in the last section.

2. Description of the ILP systems.

As there are a lot of different methods to handle the hardware features in the IPET approach, we present in this section the variant our survey is based on.

2.1. Constraints related to the task control flow

The task control flow analysis as presented in [10] involves representing the analyzed program by a Control Flow Graph (CFG), a directed graph whose nodes are Basic Blocks (BB). A BB is a sequence of instructions with exactly one entry point and one exit point (i.e. no jump in or from the middle). An edge connects two BB if the control can pass directly from the predecessor BB to the successor BB (which can happen if the two BB are in sequence, or in case of a branch, a function call/return, etc).

For each BB and each edge, an ILP variable is created, representing the number of times we execute the BB or take the edge. We name x_i the variable associated with basic block i, and name $e_{i,j}$ the variable associated with edge going from basic block i to j. To represent the structure of the CFG, structural constraints are created for each BB: the sum of execution count for all incoming edges must be equal to the sum of execution count for all outgoing edges, and to the execution count of the basic block (example for BB B_i: $\sum e_{*,B_i} = x_{B_i} = \sum e_{B_i,*}$).

An additional set of constraints is used to represent the loop bounds. A loop can be identified by a loop header L. The edges ingoing L can be divided into *entry edges*, and *back edges*. The entry edges are entering the loop, while the back edges correspond to edge allowing to perform a new iteration. The iteration bound for a loop being N means that the loop is executed at most for N iterations every time it is entered. This is matched by the constraint: $\sum backEdges \leq \sum entryEdges \times (N - 1)$. The WCET is computed by maximizing an objective function that contains a term $t_{B_i} \times x_{B_i}$ for each basic block. Therefore the resulting WCET is $max(\sum t_{B_i} \times x_{B_i})$.

2.2. Instruction Cache modeling constraints

C. Ferdinand et al. presents in [7, 6, 15, 1] a method to handle the instruction cache in the IPET approach. This article proposes a method to predict by abstract interpretation the behavior of set-associative instruction caches with a Least Recently Used (LRU) replacement policy. Thanks to three distinct analyses, the *Must*, *the May*, and the *Persistence* analyses, it categorizes the basic blocks[1] according their behavior in the cache as *Always Hit*, *Always Miss, Persistent,* or *Not Classified. Always Hit* and *Always Miss* means that the access to the cache block results ever, respectively, in a cache hit or in a cache miss. *Persistent* means, intuitively, that the first block access is a miss and the next accesses give hits. *Not Classified* means that we do not know what will happen to the block when it is accessed.

Former approach of C. Ferdinand was only considering May and Must analyses and was using loop unrolling in order to cope with a limited form of the persistence, that is, miss at the first iteration and hit in the following iteration. In [13], and later in [2], an improved approach for dealing with Persistence is proposed. In these papers, it is proposed a new form of Persistent category that takes as parameter the loop that is associated with the Persistent instruction: if the block B is categorized as Persistent at loop L, it means that L is the outer-most loop such that B can not be replaced from the cache within L. That is, L and its inner loops does not wipe out the cache block containing B and the Persistent block cause at most one miss each time the loop L is entered.

To integrate this approach with IPET, the number of hits and miss for cache blocks is represented by the variables x_i^{hit} and x_i^{miss}. In all cases, we assert that $x_i^{hit} + x_i^{miss} = x_i$. The most straightforward constraints are derived from the *Always Hit*, resp. *Always Miss*: $x_i^{miss} = 0$ (resp. $x_i^{hit} = 0$). As the $Persistent$ category means that the block will cause a miss at most each time the related loop L is entered, we can safely generate the constraint $x_i^{miss} <= \sum e_{*,L}$ (where $e_{*,L}$ represents the entry edges of the loop L).

2.3. Constraints for the CPU pipeline

In the original article of Li and Malik [10], the execution time of a basic block is viewed as a constant time produced, for example, by measuring the basic block execution time on a real processor. In [4], Engblom proposes to use simulation to compute the basic block execution times and to refine them on pipelined processor according to the neighbouring basic blocks. Yet, this approach is not safe if the processor exhibits “timing anomalies” or “long time effects”.

We have preferred the approach presented in [14] that allows to take into account the basic block context modeled from a set of parameters. The execution time can be expressed as a function of these parameters. This method is based on [9], which expresses the execution pattern of a basic block by an execution graph. This graph conveys the precedence constraints between instructions, and analyses it to derive the basic block execution time considering pessimistic assumptions about the execution context.

If the method is simply used to compute the execution time of a basic block by considering the union of possible contexts, no constraints are generated: the execution time of the basic block is modified according to the pipeline effects, and this impacts only the objective function. However, for more precision, it is possible to consider the basic block predecessors: if a basic block has several predecessors, then due to pipeline effects the basic block execution time is different, depending on the previously executed predecessor. In this case, the differences in execution time are associated to the in-edges by additional terms in the objective function.

[1]For the sake of clarity in the paper, we consider that the basic blocks are split according to cache block boundaries (allowing us to have a single category for each basic block), but our implementation handles regular basic blocks.

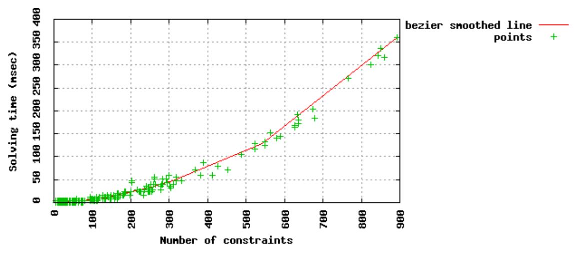


Figure 1. lp_solve solving time

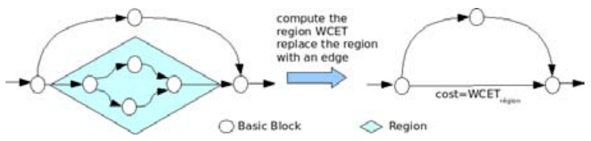


Figure 2. region building

For example, if a basic block B has two predecessors P_1 and P_2, then the terms $diff_{P1,B}. \times e_{P1,B}$ and $diff_{P2,B} \times e_{P2,B}$ are added to the objective function, where $diff_{P1,B}$ and $diff_{P2,B}$ represent the execution time of B when the control flow comes from P_1 and P_2.

2.4. ILP solving performance problems

The last few sections have shown that a lot of variables and constraints are required to compute the WCET. To sum it up, in the control flow constraints, we need a variable for each basic block and for each edge, resulting in two constraints for each basic block. Moreover, we need one more constraint for each loop header. To handle the cache, we need two additional variables and constraints (one for the category and one for $x_i = x_i^{hit} + x_i^{miss}$) for each basic block.

Finally, the use of the contextual execution graph requires only to add terms to the objective function. For a task with n_N basic blocks, n_L loops and n_E edges, usually $n_E > n_N$, we produce an ILP system with $3n_N$ variables and $5n_N + n_L$ constraints). With tasks of several thousands of basic blocks, this makes huge system to resolve and things become worse with the loop unrolling that duplicates constraints for block in nested loops ($\times 2^n$ for n nesting levels).

As the basic approach to resolve ILP systems has a super-linear complexity relatively to the constraint number, the resulting systems are very costly to solve, and takes a large part of the WCET computation time. This is illustrated in Figure 1 that shows the measured execution time for a collection of WCET computation on the lp_solve solver [11]. We have performed our measures on 100 pieces of code coming from parts or whole programs from the Mälardalen benchmark [12]. One may notice that it would be efficient to split the ILP system into subsystems smaller enough (about 100-300 constraints) because the sum of their computation times would be much lesser than the computation of the whole system. Next section introduces a method to approach such a solution.

3. Our approach

The main constraint in splitting the problem into smaller ILP subsystems is to keep the same WCET when the ILP systems are computed separately (our approach does not cause loss of WCET precision). The idea is to isolate sub-systems such such that (1) their local WCET is a constant and (2) the subsystem can be replaced in the global WCET by the product of the local WCET and of the number of times it is executed. Therefore, the subsystem isolation depends on the structure of the system, i.e. on the constraints generated to model the control flow and architecture effects.

3.1. Single-Entry Single-Exit regions

The structural constraints, for each basic blocks, relates the sum of incoming and outgoing edge count to the basic block execution count. We can observe that if a region of the CFG is connected to the outside by one entry edge, and one exit edge, the only constraints connecting region blocks with blocks outside the region are the structural constraints at the entry edge successor (BB_{entry}), and at the exit edge predecessor (BB_{exit}), whose form are $BB_{entry} = entryEdge$ and $BB_{exit} = exitEdge$. Furthermore, since we have a region with a single entry and single exit, all control flow entering by the entry edge leaves by the exit edge such that $BB_{entry} = BB_{exit} = entryEdge = exitEdge = x_{region}$, with x_{region} representing the execution count of the whole region.

Therefore, all variables representing execution count of basic blocks in the region depends ultimately only on x_{region}. Thus we can compute locally the WCET of the region, assuming $x_{region} = 1$ and taking into account only structural constraints and objective function terms corresponding to the region blocks. When the main program is analyzed, we can substitute a single edge for the region, remove from the whole system the constraints of the subsystem, remove from the objective function the terms depending on $x_i \in region$ and add to the objective function $x_{regionEdge} \times WCET_{region}$. This is illustrated in Figure 2.

It is important to keep in mind that it is consistent because the objective function terms that represents the region contribution to the WCET depends only (directly or indirectly) on x_{region}. In other words, it means that the region WCET is always the same, regardless of the execution context. Those single-entry single-exit regions are called *SESE regions* and surveyed in [8].

3.2. Handling CPU pipeline analysis

We consider here the pipeline analysis method already discussed in section 2.3. This method can compute each basic block execution time for the union of all possible contexts. In this case, the execution time of each basic block is independent, and the WCET of the region is the same in every context. Only the objective function needs to be modified: each term $t_i \times x_i$ is adjusted (t_i is set to the execution time of basic block i computed by the pipeline analysis). In this case, no additional problem appears, and the SESE region approach presented in the last section can be used as is.

The CPU pipeline analysis method can also compute the basic block execution depending on the predecessors. In this case, the objective function is enhanced by $diff_{i,B}. \times e_{i,B}$ terms for each predecessor of each basic block B. The unique basic block whose predecessor may be outside the region is the entry basic block (that is, the sink block of the region entry edge). Because the regions are single-entry single-exit, the entry basic block has only one predecessor, and thus, one possible execution time. Therefore, in this case, too, no additional problem appears. To sum it up, the CPU pipeline handling does not cause problems with our approach, provided we limit ourselves to (1) a single time for each basic block or (2) a time dependent on the predecessor only. Experimental results found in [14] shows that WCET tightness is pretty good with a context of only one predecessor.

3.3. Handling cache related constraints

The cache categorization method by C. Ferdinand et al [7, 6, 15, 1] computes categories such as *Always Hit*, *Always Miss*, *Persistent*, and *Not Classified*, and generates ILP constraints based on those categories as described in section 2.2. In order to know if the cache categorization raise any issue

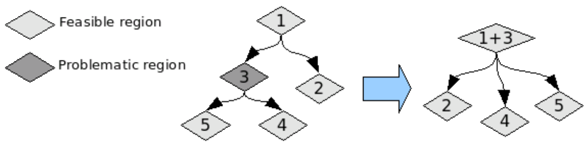


Figure 3. From PST to Feasible Tree

regarding our region partitioning approach, we need to study the ILP constraints generated by each of the possible categories (except for *Not Classified*, which does not generate any constraints).

The *Always Hit* and *Always Miss* categories means that the block causes, respectively, always a hit or a miss. Therefore, the generated constraints, $x_i^{hit} = x_i$ or $x_i^{miss} = x_i$, and the objective function contribution, $x_i^{hit} \times t_i^{hit}$ and $x_i^{miss} \times t_i^{miss}$, does not cause any difficulty because their value is proportional to x_i (either equal, or zero) that, in turn, in derived from x_{region}. So the terms of the objective function to handle the cache may be embedded in $x_{region} \times WCET_{region}$. The loop unrolling approach does not induce any problem as it only duplicates some parts of the code.

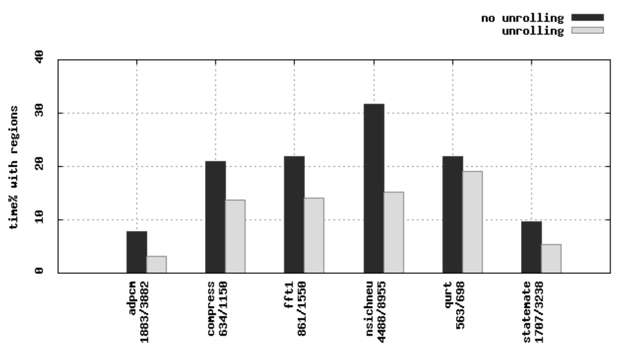
The *Persistent* category have more complex constraints that link together basic blocks count variables from distant CFG parts. The *Persistent* category means that the first execution of the instruction may result in a miss, but all the subsequent executions will result in hits. The corresponding constraint has the form $x_i^{miss} <= \sum e_{*,H_L}$ (H_L is the loop header of L, the loop associated with the *Persistent* block). If the loop header is outside the region, then x_i^{miss} depends on something that can not be derived from the execution count of the region, x_{region}. In other words, for each execution of the region, the hit/miss status of the *Persistent* basic block depends on the presence of the block in the cache at the region entry. Therefore, the region WCET is different depending on the context.

The Persistent category show us that the constraints induce specific limitations to the local computation of region WCET: a region can be computed independently if for any *Persistent* basic block, the associated loop header is also in the region, else the region is infeasible. In a more generic way, the region partition is driven by the nature of the constraints and the objective terms applied to the basic blocks of the region. A region is feasible if (1) the objective function terms tied to the region blocks are proportional to the number of times the region is executed, x_{region}, and (2) any constraint containing variables tied to the region blocks contains only variables tied to the region blocks.

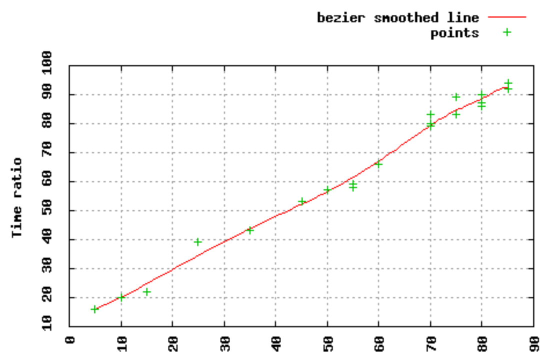
3.4. The Program Structure Tree

We have seen that hardware effects (cache) may disrupt our approach by making some regions infeasible, thus slowing the overall WCET computation. So, to improve the efficiency of our method, we propose to compensate the apparition of non-feasible regions by organizing the integration of local WCET in a hierarchy of computations. This forms a tree called *Program Structure Tree* (PST) [8]. This paper introduces the concept of Single-Entry Single-Exit (SESE) regions, a sub-graph of the CFG with one entry edge and one exit edge. It defines a *canonical* SESE region as a SESE region that does not contain any other SESE region that shares its entry or its exit edges. It is then shown that the canonical SESE regions can be structured in a PST that can be computed in linear time.

The PST is not usable as is, because the hardware effects makes some regions infeasible. The PST building must be modified to remove the infeasible regions, as described in the Figure 3. The PST is visited from the leaves to the root. When an infeasible region is found, it is removed by moving



(a) time gained with lp_solve



(b) sensitivity to random dependencies

Figure 4. Experimental results

its basic blocks and its children regions to its parent region. The result is a tree containing only feasible regions. Finally, the whole WCET is computed by performing a bottom-up visit of the PST, computing each region WCET once all its children are processed and by reporting the children WCET into the parent region system.

4. Evaluation

To validate our approach, we have experimented with it using OTAWA [3], our WCET computation framework. The target architecture features a simple pipelined processor (handled by contextual execution graph) and a 4-way set-associative instruction cache (handled by cache categorization). The measurements have been done on a subset of big-enough Mälardalen benchmarks [12]), with the lp_solve solver [11]. A “big-enough” benchmark means that the generated ILP system provides enough space to extract regions (generating more than 300 constrains).

4.1. Comparison of analysis times

For each selected test of the Mälardalen benchmarks, we have computed the WCET with both the traditional and the region partitioning approaches, and measured the computation time (excluding program path and architecture effects analysis that remains unchanged in our approach). In the case of the region partitioning approach, we have taken into account the time needed to do the partitioning (PST building, identification of infeasible regions, etc).

On average, with our approach the analysis is 6.5 times faster, and results exactly in the same computed WCET. Details can be found on figure 4(a), which gives for each test the percent of the ratio $\frac{time_{new-approach}}{time_{old-approach}}$ (for example, 33% means that we go approximately 3 times faster), with and without loop unrolling.

4.2. Results with random dependencies

To evaluates the limits of our approach, we have also experimented measurement with additional random dependencies. These dependencies generalize the concept of long range dependencies as generated by First Miss blocks in the cache analysis: such a block is linked to the loop header by a constraint. In other words, the dependency from a source basic block BB_1 to a sink basic block BB_2

represents the fact that we need information about BB_1 number of executions to know the execution time of BB_2.

This property could also be applied on any analysis involving categorization, not just cache behavior prediction. As we have no WCET algorithm to produce these dependencies, we have used a random generator. The generated dependencies prevents the feasibility of some regions and stresses our algorithm.

The figure 4(b) shows the impact of these random dependencies on the analysis time. The X axis represents the ratio in percent of the basic blocks affected by a dependency, while the Y axis represents the mean analysis time over all tests for the same Mälardalen benchmarks as in previous section. The results shows that, even if almost every other basic block is affected by a dependency, we still have some time gain.

5. Conclusion

This paper proposes an approach to alleviate the problem of high analysis time for WCET computation with the IPET method by partitioning the program into smaller regions that can be computed independently. We have presented the general principle of our approach (building the PST and visiting it in a bottom-up fashion to compute the WCET of the program) in the trivial case where we do not take into account architecture effects. We have then identified the limitations of our approach in more realistic conditions, including cache behavior prediction, CPU pipeline analysis and even random generation of dependencies between basic blocks. The experimentation conducted in several situations shows a significant time gain.

In future works, we plan to support more architecture effect modeling like the branch prediction unit or the data caches. Although the random dependency generation seems to show that the approach remains efficient, it would be interesting to stress the algorithm with more complex constraints as found in Cache Conflict Graph for example. Another way of experimentation concerns the validation of the method with other industrial or publicly available ILP solvers. Any solver based on the dual simplex algorithm should exhibit some improvements in computation time but additional ILP techniques may offset the time spent in our analysis.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[2] C. Ballabriga and H. Cassé. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2-4 July 2008.

[3] H. Cassé and P. Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software*, 25-27 December 2005.

[4] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. In *Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC)*, April 1999.

[5] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9):1104–1122, September 2005.

[6] C. Ferdinand. A fast and efficient cache persistence analysis. *Technical report, Universitat des Saarlandes*, Sept. 1997.

[7] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.

[8] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[9] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[11] lp_solve ILP solver, http://lpsolve.sourceforge.net/.

[12] Mälardalen benchmarks, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[13] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, pages 209–239, 2000.

[14] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3):109–128, 2007.

[15] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

INFER: INTERACTIVE TIMING PROFILES BASED ON BAYESIAN NETWORKS [1]

Michael Zolda[2]

Abstract
We propose an approach for timing analysis of software-based embedded computer systems that builds on the established probabilistic framework of Bayesian networks. We envision an approach where we take (1) an abstract description of the control flow within a piece of software, and (2) a set of run-time traces, which are combined into a Bayesian network that can be seen as an interactive timing profile. The obtained profile can be used by the embedded systems engineer not only to obtain a probabilistic estimate of the WCET, but also to run interactive timing simulations, or to automatically identify software configurations that are likely to evoke noteworthy timing behavior, like, e.g., high variances of execution times, and which are therefore candidates for further inspection.

Keywords: Bayesian networks, embedded systems, hardware modeling, measurement-based execution time analysis, software modeling, probabilistic modeling, profiling, real-time systems

1. Introduction

With the increasing number of embedded computer systems in everyday life applications, like cars, digital entertainment systems, or mobile phones, knowledge about the real physical behavior of such systems is becoming more and more important. In particular, when a computer system is embedded in a real physical process, like in an engine control system, or a digital media stream decoder, knowledge of its timing properties becomes crucial.

Despite the notable advances in timing analysis of embedded computer systems, performing a WCET analysis of state-of-the-art systems is becoming more difficult, as intricate processor features, like caches, pipelining, and branch prediction, trickle down from the desktop and server to the embedded processor domain. As a result, such systems are becoming too complex for a complete, detailed analysis.

Whenever we have to reason about systems the complexity of which prohibits us from explicitly dealing with each special case, but the details are too important to simply ignore them, it is common and acknowledged practice to use models that *summarize* the impact of exceptions.

Probabilistic network models [10] provide a theoretically sound framework for summarizing complex relationships as probabilistic dependencies. Besides providing mechanisms for simulating the model's behavior under various user-defined scenarios, probabilistic network models offer a better traceability

[2]Institut für Technische Informatik, Technische Universität Wien, Treitlstraße 3/182/1, A-1040 Wien, Austria, e-mail: michaelz@vmars.tuwien.ac.at

[1]The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13.

of effects, compared to traditional regression models.

We propose an approach that uses Bayesian networks to model program execution times of software-based embedded systems. The structure of our networks is based on the structural properties of the soft- and/or hardware under test. The network is subsequently parameterized with empirical data obtained from run-time measurements on the actual target hardware.

2. State Machine Models

2.1. Reactive Systems

Depending on their method of operation, embedded computer systems can be classified into *reactive systems* and *transformative systems*. Whereas a reactive system keeps on running continuously, interacting with its physical environment, a transformative system is characterized by a "one-shot" mode of operation: the system starts by taking all its inputs, performs some calculation, and terminates with some output. Transformative systems are popular basic building blocks for more complex systems, where they are invoked periodically by a scheduler. If the transformative system is part of a (hard or soft) real time system [5], then it must fulfill certain previously specified timing constraints. Not meeting those timing constraints is considered a system failure.

We consider software-based transformative systems. Even though there have been various proposals to make such systems time-predictable by design, software that was written by applying traditional programming techniques, and/or which is running on modern processors has a highly unpredictable temporal behavior [4].

When we take a mathematical point of view on such a system, we essentially face an extremely complex state machine, where each state corresponds to an intricate hardware configuration, and where the transitions correspond to the change from one such configuration to another. Since each change between states corresponds to a real physical processes, we can associate an execution time with each transition. At this level of detail, the system is deterministic, and the execution time for each transition is cycle-accurate.

Unfortunately, we usually cannot describe a real transformative system in all details, for various reasons, like, e.g., inadequate detail of the available hardware specification, or sheer size. So we have to abstract our state machine model (SMM).

We consider three forms of abstraction, which are performed in the given order: *State elimination*, *Existential abstraction* and *segment abstraction*.

2.2. State Elimination

State elimination is achieved by removing states from the SMM. When a state s is removed, each of its incoming edges is redirected to the unique successor node s'[1]. Subsequently, the transition between s and s' is dropped, after adding its associated execution time to each of the redirected edges.

[1]The uniqueness is guaranteed by our premise of determinism of the SMM. From this uniqueness, it also follows that the SMM is free of cycles, because we are modeling a transformative system that is supposed to terminate with some output.

2.3. Existential Model Abstraction

Existential abstraction is achieved by collecting the concrete states of the SMM into sets, and viewing these sets as the abstract states of an *abstract state machine model (ASMM)*. The transitions of the ASMM are induced by may-semantics: There is an abstract transition between two abstract states of the ASMM, iff there is at least one concrete transition between two concrete states of the underlying SMM. Accordingly, the execution times of all the concrete transitions are collected into a multi-set with a corresponding may-semantics.

For software written in imperative programming languages, we usually consider *basic blocks* and their static control flow structure. In that case, transitions are identified with basic blocks, and abstract states are identified with basic block numbers.

Figure 1(a) gives an example source code written in the C programming language. Figure 1(b) shows the structure of the corresponding ASMM.

2.4. Segment Model Abstraction

Segment Abstraction is another form of abstraction, normally performed after existential abstraction. Whereas in existential abstraction we collect states into abstract states, in segment abstraction we collect paths into *segments*.

A segment is characterized by its entry/exit interface. It essentially binds together all paths that start at the entry interface (which is given as a collection of states) and end at the exit interface (another collection of states).

The utility of segments is threefold:

Handling of path explosion. The execution time of an individual basic block generally depends on the execution history. As a consequence, the execution times of individual basic blocks are not composable without loss of information. If the executions times are, for example, given as probability distributions, combining them through a convolution operator might lead to over- and underestimation of the probabilities of certain execution times. Even more importantly, in the measurement-based approach individual measuring of basic blocks may totally miss execution times that depend on a rare system state that is only reached through certain execution histories.

For maximal accuracy w.r.t. the ASMM, in the measurement-based approach, we would have to measure the execution time of each ASMM path. However, the number of ASMM paths is typically prohibitively large.

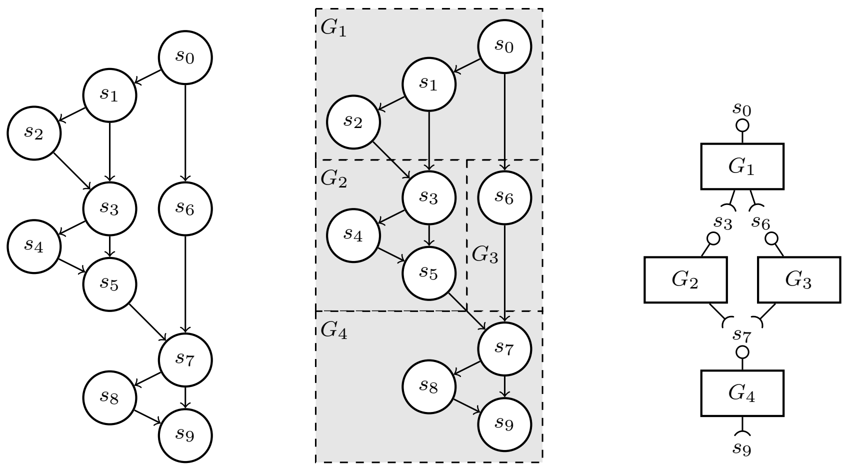
Segments are used to specify restricted measurement/coverage regions. They should be chosen such as to contain a limited number of paths, thus alleviating the path explosion problem. However, the paths within the segments should not be too short, such as not to sacrifice too much history-dependent information. A good segmentation algorithm should balance these two opposed goals.

Dependencies across segments can be captured/modeled by the Bayesian network model.

Obtaining accurate measurements. When we want to obtain the execution time information through measurement, we face the problem that some measurement methods are intrusive in the sense that the

```
s0: if(m==0) {
s1:   if(y!=0)
s2:     m=x%y;
s3:   if(m==0)
s4:     z++;
s5:   k++;
    } else {
s6:   f=1;
    }
s7: if(m==0) {
s8:   f=0;
    }
s9:
```

(a) Example source code



(b) Abstract state machine model (ASMM) (c) ASMM with segmentation (d) Abstract view on the segmentation

Figure 1. Illustration of an abstract state machine model (ASMM) and its segmentation

act of measuring affects the result. A typical example of this is measuring using software instrumentations. Such instrumentations affect the hardware state and may therefore also the affect the execution time of subsequent code. Secondly, some measurement methods can only provide limited accuracy, e.g. if they employ a timer with low granularity.

Both effects can be alleviated by avoiding the measuring of short sequences of instructions. Segments are used to specify measurement spans of a minimal length that is specific to the applied measurement method.

Logical Structuring. We do not only want to model the overall execution time behavior of the entire program, but also want to get more insight into the timing interactions within the program.

Segment are used to structure the program into smaller parts whose execution time behavior is made visible to the user.

For example, the engineer might consider a small loop as basic unit of functionality. Then this loop would be a candidate for a segment.

A *segment* is an ordered pair $\langle E, X\rangle$, where E is a set of *entry states*, and X is a set of *exit states*. Semantically, E is related to X through the segment $\langle E, X\rangle$, iff, for each state in E, there is a directed path to some state in X that does not touch any other state in X.

The set of *associated paths through* a segment $\langle E, X\rangle$ is the set of all paths[2] $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \dots \xrightarrow{l_{n-1}} s_n$, such that $s_0 \in E$, $s_n \in X$, and $s_i \notin X$, for any $0 < i < n$ and any $n > 0$.

Figure 1(c) indicates the partitioning of the ASMM from Figure 1(b) into four segments. For example, segment G_1 is the segment with $E = \{s_0\}$ and $X = \{s_3, s_6\}$. We can reach s_3 from s_0, e.g. via s_1, without touching s_6, and we can reach s_6 directly from s_0. The paths associated through G_1 are $s_0 \to s_1 \to s_2 \to s_3$, $s_0 \to s_1 \to s_3$, and $s_0 \to s_6$. Figure 1(d) provides a more abstract view of the

[2]We use labels to distinguish parallel edges. If there are no parallel edges, we can omit them.

segments and their interconnection. Note how the entry and exit states act as “connectors” between segments.

The idea behind segments is to abstractly and implicitly specify the set of paths that can be taken through a given region simply as an entry/exit interface. We then associate, with each segment, the multi-set of execution times of all its associated paths.

A segment is an over-approximation of the underlying subgraph of the SMM, in the sense that it summarizes all possible paths through that subgraph, and thus all possible path execution times. Segments should thus be seen as the primitives of a program with which we can associate meaningful timing information.

Segment abstraction can be performed by iteratively replacing subgraphs of the ASMM with matching segments, until all transitions have been collected into segments.

For a given ASMM, the choice of segments is not unique. We do not, at this moment, provide a concrete algorithm that incorporates all the requirements on segments that were discussed in this section.

3. Probabilistic Modeling

3.1. Bayesian Network Essentials

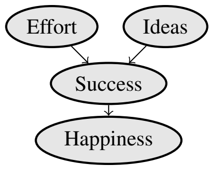
In probability theory, we consider *random experiments*, i.e., experiments with an outcome that is governed by some indeterministic mechanism. The possible outcomes of such an experiment are called *events*. To express a certain degree of confidence that a certain event will occur, real numbers from the interval $[0 \dots 1]$ are used. Greater values indicate greater confidence; a value of 1 indicates absolute confidence that an event will occur, 0 indicates absolute confidence that an event will fail to occur, 0.5 indicates total indifference about the occurrence of an event. These real numbers are called *probabilities*.

A *random variable* is a variable whose value depends on the outcome of a random experiment. Through this dependency, random variables inherit probability values from the probabilities of their underlying events. Consequently, constraints over random variables can themselves be viewed as events.

When performing probabilistic reasoning, we always consider a specific *probabilistic model* that describes a set of random variables, as well as their qualitative and quantitative interconnections.

The *conditional probability* $P(X{=}x \,|\, Y_1{=}y_1, \dots Y_n{=}y_n)$ designates the probability that variable X obtain the value x, given the *a-priori knowledge* that Y_i obtain the value y_i, for $1 \leq i \leq n$. Note that the order of the conditions is irrelevant, but that generally $P(X{=}x \,|\, Y{=}y, \dots) \neq P(Y{=}y \,|\, X{=}x, \dots)$. The *unconditional probability* $P(X{=}x)$ designates the probability that variable X obtain the value x, given no further information about the outcomes of any other variables in the model.

Two variables X and Y are *conditionally independent given a set of variables* $Y_1, \dots, Y_n$, iff $P(X{=}x \,|\, Y{=}y, Y_1{=}y_1, \dots Y_n{=}y_n) = P(X{=}x \,|\, Y_1{=}y_1, \dots Y_n{=}y_n)$. Intuitively speaking, the outcome of Y does not influence the outcome of X, under the given a priori knowledge. Conditional independence



	Success	¬Success
Happiness	0.6	0.3
¬ Happiness	0.4	0.7

	Effort		¬Effort	
	Ideas	¬Ideas	Ideas	¬Ideas
Success	0.7	0.4	0.3	0.2
¬ Success	0.3	0.6	0.7	0.8

Effort	0.5
¬ Effort	0.5

Ideas	0.5
¬ Ideas	0.5

Figure 2. Example of a bayesian network

is a symmetric relation.

Bayesian network (BN) are a representation mechanism that can express the important class of *causal probabilistic models*. In a BN, the model's variables are expressed as nodes of a graph. For each variable X, the BN has a conditional probability table (CPT) that specifies the conditional probabilities $P(X{=}x \,|\, Y_1{=}y_1, \ldots Y_n{=}y_n)$ over all possible combinations of outcomes of its parent variables $Y_1, \ldots Y_n$.

The connection between a BN and the represented probabilistic model is completed by the following formal property: Any variable X is conditionally independent of all its non-descendants, given its parents $\mathbf{Y} = \{Y_1, \ldots Y_n\}$, and no subset of $\mathbf{Y}$ satisfies this condition [10]. Intuitively speaking, each *direct causal dependency* between variables of the model implies a corresponding directed arc in the BN.

Figure 2 depicts a simplified Bayesian network model of success. The variables of this simplified model—effort, ideas, success, and happiness—are all binary: either you are happy or not. The probability of success depends on both, effort and ideas. Effort indirectly affects happiness via success, but there is no *direct* causal influence. Therefore, if you are definitely a successful person, making a change in effort won't affect your 60% probability of happiness: Happiness is conditionally independent from effort, given success. If there was a direct connection between effort and happiness, we would have to add an extra arc between them.

Note how the conditional probability tables specify, for each node, the probability of each outcome, for all combinations of outcomes of their parent nodes. For effort and ideas, which have no parents in our model, we need to specify the unconditional a priori probabilities. The uniform 50-50 distributions are appropriate to express complete ignorance about the unconditional likelihood of effort or ideas.

3.2. Using Bayesian Networks for Simulation

The salient feature of a Bayesian network is its direct applicability for simulations. The network constitutes a complete probabilistic model of the variables in its domain and can, amongst other things, be used to solve "what-if?" scenarios. A simple simulation on a BN consists of the following steps:

1. The user provides *hypothetical evidence* $Y_1 = c_1, \ldots, Y_n = c_n$ for some selected nodes of the network[3], i.e., he fixes the value of some variables to one of their possible outcomes.

[3]Note that this hypothetical evidence is provided as part of a query. The user is not required to provide such information during the model construction. Also, it is possible to have queries without any hypothetical evidence (where $n = 0$).

2. A *belief update* is performed on the network[4]. During this operation, the conditional probabilities $P(X = x|Y_1 = c_1, \ldots, Y_n = c_n)$ are evaluated for all network variables X. Note that $c_1, \ldots, c_n$ are constant values. The conditional probabilities $P(X = x|Y_1 = c_1, \ldots, Y_n = c_n)$ thus represent the probability distribution of random variable X under the *user-defined scenario* $Y_1 = c_1, \ldots, Y_n = c_n)$.

3. The user reads the probability distributions of his interest from the model.

For illustration, consider, once again, the Bayesian network from Figure 2. Assume that the user wants to perform the *diagnostic query* “How likely is it that a happy person is putting in some effort?” By setting the evidence of the happiness node to true and running a belief update over the network, the user can obtain the numeric answer, which is “55%”, i.e., slightly higher than the corresponding a priori likelihood of “50%”. A detailed explanation of the corresponding calculations is out of the scope of this work, but for an intuive explanation, consider that fixing happiness to “true” leads to an increase of the likelihood of success, via the direct link between those nodes. The increased likelihood of success, in turn, yields an increase in the likelihood of both, effort and ideas.

We are convinced that such a mechanism for performing simulations of user-defined “what-if?” scenarios is a highly desirable and useful tool in the hands of an engineer who is performing a timing analysis of a given system.

3.3. Probabilistic Interpretation of Abstraction

A segment (E, X) summarizes the timing behavior of all SMM paths from E to X by collecting the execution times of these paths in a multi-set. Interpreting relative frequencies as probabilities, we obtain, for each segment, a random variable on the possible execution times.

The random variables for different segments are generally not conditionally independent. Rather, they show some degree of correlation; a result of the fact that the concrete SMM path taken through a segment during an actual run of the software depends on the concrete SMM state through which the segment is entered, and thus on the concrete SMM path that was taken through previous segments.

At our level of abstraction, the correlations between the execution times of basic blocks can be viewed is being established via a “hidden” information channel. For example, an instruction cache can act as a mediator between the timing variables of two basic blocks that share a common cache line, creating a probabilistic dependency between them.

Once we have identified such information channels, we can create a Bayesian network model that incorporates the corresponding probabilistic dependencies.

3.4. Deriving the Network Structure

Besides performing the segmentation of the ASMM, we also have to identify the relevant dependencies between segments.

Let G be a segment. The *context set* $cs(G)$ of G is the set of all segments on which G *causally* and *directly* depends.

[4]Various different algorithms that perform this task have been proposed and implemented [9, 6, 3, 14].

For our application, the restriction to *causal* dependencies means that $cs(G)$ can only contain ancestor segments (w.r.t. the flow of control) of G. The restriction to *direct* dependencies means that transitive dependencies are not modeled explicitly.

Technically, the variables associated with the context set of $cs(G)$ will later form the Markov boundary of the variable associated with G.

The identification of context sets should generally be done in parallel to identifying segments, as both choices depend on each other.

Given a segment G, how can we determine the context set $cs(G)$ of G?

It is important to note that we cannot provide a feasible method for automatically identifying all dependencies of G. Rather, we propose a best-effort approach that yields a good approximation of $cs(G)$ by considering *candidate segments* that are likely to exert a strong influence on G.

Our primary source for candidate segments is our abstract knowledge about the hardware architecture and software semantics.

For example, if we have some knowledge about the instruction cache and memory layout of our target architecture, then we might be able to identify segments that are in conflict through the sharing of a common cache line.

Another example is the consideration of pipelining effects over segment borders: in a pipelined architecture, a segments potentially depends on its immediate predecessors through the shared pipeline state at the common segment boundary.

Our third example is the analysis of control flow dependencies. In the following we give a sketch of how we can identify candidate segments for $cs(G)$ through the use of use-definition chains.

Let x be a program variable that is used inside the condition of a control flow statement within segment G. Since the value of x can influence the flow of control in G, it also has a potential influence on the execution time of G. Next, consider the corresponding assignments of x (which can be easily obtained by static program analysis). If such an assignment of x is contained in at least one branch of a control flow statement St of segment G', then G depends on G' by way of the “hidden” common dependencies of both segments on the condition of St.

3.5. Classifying Execution Times

In a Bayesian network, we need to specify, for each variable X, the conditional probabilities w.r.t. all parent variables. If we have n possible outcomes (here: different execution times) for each segment, then the CPT for a segment with a context set size of m requires n^{m+1} entries. It is thus clear that we have to limit both, the size of the context set for each segment, and the number of outcomes for each variable.

The size of the context set of a segment can be reduced by considering only the strongest dependencies. The strength of a dependency can, for example, be judged by the *measure of mutual information* [13] of corresponding variables.

On the other hand, the number of outcomes for a variable can be reduced by classifying values. For example, if the possible execution times for segment G are in the range of 100 to 750 microseconds, then we might summarize the possible outcomes as intervals $[100, 250)$, $[250, 500)$, $[500, 750]$, plus a special outcome "null" that represents the situation where a segment is not executed. A reasonable classification is crucial for obtaining a significant probabilistic model.

3.6. Parameterizing the Network with Measurements

Once we have identified segments and their associated context sets, we have to parameterize the network with appropriate conditional probabilities of execution times. As mentioned in Section 3.3, our probabilities are based on relative frequencies.

We obtain the relative frequencies of execution times by performing measurements on the real computer system. The main advantage of our measurement-based approach is that we obtain our timing data from the actual system, which incorporates all the numerous and possibly peculiar implementation details and quirks of the future production system.

To parameterize our network, we need to obtain, for each segment G, the *conditional relative frequencies* $f(T = c \,|\, T_1 = c_1, \ldots, T_n = c_n)$ of execution times, where $T, T_1, \ldots, T_n$ are random variables representing the execution times for the segment G, and its corresponding context segments $\{G_1, \ldots G_n\} \in cs(G)$, and where $c, c_1, \ldots, c_n$ are the corresponding classes of execution times. This can be achieved as follows:

1. Generate test data vectors $d_1, \ldots, d_m$, that force a flow of control through G. This can be done by harnessing techniques like random test data generation and model checking [11].
2. For each test data vector d_i, $1 \le i \le m$, measure the execution times $t_{i,G}, t_{i,G_1}, \ldots, t_{i,G_n}$ of $G, G_1, \ldots, G_n$. Available options for this step are the use of intrusive techniques like static source code instrumentation, or non-intrusive techniques like timing trace generation with hardware trace probes.
3. Obtain the *joint absolute frequencies* as

   $$F(c, c_1, \ldots, c_n) = |\, \{ i \,|\, t_{i,G} \in c,\ t_{i,G_1} \in c_1,\ \ldots,\ t_{i,G_n} \in c_n,\ 1 \le i \le m \} \,|,$$

   and subsequently the conditional relative frequencies as

   $$f(c \,|\, c_1, \ldots, c_n) = \frac{F(c, c_1, \ldots, c_n)}{\sum_{x \in domain(T)} F(x, c_1, \ldots, c_n)}.$$

To obtain representative frequency distributions for the segment, the test data vectors $d_1, \ldots, d_m$ should throughly cover the timing behavior of G. Since a full coverage of all possible SMM paths is usually infeasible, we confine ourselves to up to k random samples per ASMM path, i.e., we try to achieve full path coverage of the abstract model within the segment and try to generate up to k unbiased measurements per path. Full ASMM path coverage can be achieved by applying techniques like random test data generation and model checking [11].

T_{S_1}		10ms			11ms			null		
T_{S_2}		20ms	21ms	null	20ms	21ms	null	20ms	21ms	null
	30ms	1	0	0	0	0	0	0	0	0
	31ms	0	1	0	1	0	0	0	0	0
T_S	32ms	0	0	0	0	1	0	0	0	0
	null	0	0	0	0	0	0	0	0	1
	inconsistent	0	0	1	0	0	1	1	1	0

Table 1. Example CPT that relates the execution time of two sequential segments S_1 and S_2 with the execution time of their corresponding super-segment S

3.7. Multiple Layers of Abstraction

The modeling we have described so far produces Bayesian networks that capture the running times of individual segments. However, the engineer will usually also be interested in the execution times of larger program sections, in particular the execution times of the whole program (most specifically in the overall WCET). To this end, we introduce multiple layers of abstraction to our model.

Above the basic segmentation layer, we introduce a coarser *super-segmentation* layer, such that the basic segmentation layer can be seen as a refinement of the coarse layer. The network corresponding to the super-segmentation layer is, however, not parameterized by measurements, but the values of these nodes depend functionally on their corresponding basic segments.

Such functional dependencies can by modeled as “deterministic” conditional probability tables, i.e., CPTs that contain only zeros and ones. Table 1 shows an example CPT that relates the execution time of two sequential segments G_1 and G_2 with the execution time of their corresponding super-segment G. Note the outcome “inconsistent”, which is an artifact of our modeling approach. We must include such hypothetical situations in our model to capture anomalous flows of control between segments that violate structural flow constraints [8]. However, additional consistency nodes can easily reduce the actual probability of these outcomes to zero.

The final BN model includes several layers of segment abstraction, where the top layer contains only a single segment that represents the overall behavior of the system. The corresponding variable captures the execution time of the complete system, including the empirical, probabilistic worst case outcome.

4. Related Work

Lemeire and Dirkx [7] present an approach for performance analysis of concurrent systems that is based on Bayesian networks. Whereas the structure of our networks is based on the structural properties of the system under test, the approach of Lemeire and Dirkx is based on a functional description of the system that requires system-specific knowledge. In contrast to this, our approach is generic. Eventually it should be possible to perform all steps, i.e., construction of the ASMM, segmentation, identification of candidate dependencies, and parameterization, automatically.

Bernat et al. [1, 2] present a probabilistic approach for WCET calculation where the execution time frequency distributions of different code sections are combined by one out of three combination operators. The choice of the operator depends on whether the distributions are correlated via a known joint distribution, correlated via an unknown joint distribution, or uncorrelated. The operators are used to calculate an overall frequency distribution for the whole program. Compared to this operational approach, which is targeted at the derivation of one particular *static distribution*, our approach is aimed at the derivation of an *interactive timing model* of the system under test, on which the user can perform arbitrary simulations. Obtaining the overall execution time distribution over all possible inputs (and thus obtaining a probabilistic estimate of the WCET) is only one particular use case of such a timing model.

The concept of ASMM segments that we introduce in this paper is a generalization of the concept of CFG segments introduced in [11, 12].

5. Future Work

In Section 2.4 we introduced the concept of segments and indicated their application. We have yet to provide a concrete segmentation algorithm.

In Section 3.5 we argued that we have to limit the number of possible outcomes of random variables, and proposed classification as a solution. We are currently working on a WCET-aware classification method for execution times that tries to minimize the loss of information that is relevant for WCET calculation.

In Section 3.7 we have presented a brief description of how we can include multiple layers of model abstraction into a single Bayesian network model. Further work is needed on this concept.

The Bayesian network structure that we propose in this paper models variables for segment execution times and their dependencies. We are currently developing a scheme for deriving much richer network models that expose conditions on program variables and the flow of control. These models will feature clearer and hopefully more intuitive dependency structures and richer choices for simulation.

We are planning to integrate the presented approach in the timing analysis suite that is currently being developed within the FORTAS project[5]. Within the project, we are developing a fine-grained abstract system machine model, where states will carry more information than merely a basic block number. Augmenting and adapting the presented concepts to this model will be a challenge for future work.

Also, in acknowledgment that our ideas need to be tested within quantitative experiments, we are planning to provide a working implementation of our approach within the FORTAS framework.

6. Summary and Conclusion

In this work, we have presented a probabilistic approach for modeling the execution time of software-based embedded systems that is based on the framework of Bayesian networks. The structure of our networks, which represents the conditional dependencies between execution times, is derived from knowledge about the hardware architecture and software semantics, where is the parameterization is obtain by performing measurements on the real physical system.

The salient benefit of having a Bayesian network model of the timing behavior of the system under test is its ability to let the engineer perform simulations, like, e.g., "what-if" timing scenarios. We are convinced that this ability provides a highly desirable and useful tool to the hands of an engineer who

[5]The FORTAS project is a cooperation between the Real Time Systems Group at the TU Wien and the Formal Methods in Systems Engineering Group at the TU Darmstadt, with the goal of developing a software engineering oriented timing analysis method that integrates measurement-based and formal methods. Please c.f. `http://fortastic.net/`.

is performing a timing analysis of a given system.

In particular, the querying abilities of the BN model subsume unconditional queries for the total execution time of the underlying system. Such queries return a probability distribution of total execution times. The upper bound of that distribution is a probabilistic estimate of the WCET of the system. Thus, our approach is more general, and provides a broader setting for timing analysis then pure WCET calculation.

7. Acknowledgment

We would like to thank the reviewers for providing many useful hints on how to improve the present paper and pointing out the aspects of our approach that require particular attention in our future work.

References

[1] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[2] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pwcet: a tool for probabilistic worst case execution time analysis of real–time systems, 2003.

[3] Jian Cheng and Marek J. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[4] Raimund Kirner and Peter Puschner. Obstacles in worst-cases execution time analysis. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Orlando, Florida, May 2008.

[5] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer, 1997. ISBN: 0-7923-9894-7.

[6] Steffen L. Lauritzen and David J. Spiegelhalter. *Readings in uncertain reasoning*, chapter Local computations with probabilities on graphical structures and their application to expert systems, pages 415–448. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[7] Jan Lemeire and Erik Dirkx. Causal models for parallel performance analysis. In *Proc. 5th PA3CT Symposium*, Edegem, Belgium, Sept. 2004.

[8] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[9] Judea Pearl. Fusion, propagation, and structuring in belief networks. *Journal of Artificial Intelligence*, 29(3):241–288, 1986.

[10] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann Publishers, San Mateo, CA, 1988. ISBN: 0-934613-73-7.

[11] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*, Munich, Germany, Mar. 2005.

[12] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.

[13] Yiyu Yao. *Entropy Measures, Maximum Entropy Principle and Emerging Applications*, chapter Information-theoretic measures for knowledge discovery and data mining, pages 115–136. Springer, 2003.

[14] Changhe Yuan and Marek J. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *Proc. 19th Annual Conference on Uncertainty in Artificial Intelligence*, 2003.

Merging Techniques for Faster Derivation of WCET Flow Information using Abstract Execution

Jan Gustafsson and Andreas Ermedahl
School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson, andreas.ermedahl}@mdh.se

Abstract
Static Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component in static WCET analysis is to derive flow information, such as loop bounds and infeasible paths.

We have previously introduced abstract execution *(AE), a method capable of deriving very precise flow information. This paper present different merging techniques that can be used by AE for trading analysis time for flow information precision. It also presents a new technique,* ordered merging, *which may radically shorten AE analysis times, especially when analyzing large programs with many possible input variable values.*

1. Introduction

The *worst-case execution time* (WCET) is a key parameter for verifying real-time properties. A *static WCET analysis* finds an upper bound to the WCET of a program from mathematical models of the hardware and software involved. If the models are correct, the analysis will derive a timing estimate that is *safe*, i.e., greater than or equal to the WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as bounds on the time different instructions may take to execute, as well as the program's *possible execution flows*, such as bounds on the number of times each instruction can be executed, needs to be derived. The latter, so called *flow information*, includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, dependencies between code parts, etc.

The goal of a *flow analysis* is to calculate such flow information as automatically as possible. Flow analysis research has mostly focused on *loop bound* analysis, since upper bounds on the number of loop iterations must be known in order to derive a WCET estimate. Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and possible input data values [9]. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the resulting WCET estimate. In general, a flow analysis which can take constraints on input data values into consideration, a so called *input sensitive analysis*, should be able to derive more precise

This research has been supported by the KK-foundation through grant 2005/0271. Travel support has been provided by the ARTIST2 European Network of Excellence.

flow information than an analysis which cannot [2].

One promising flow analysis method is *Abstract Execution* (AE), which is able to find precise flow information for many different types of programs [2, 9]. AE works by abstractly executing the paths which the program may execute for all its different input values. To avoid a combinatorial explosion of the number of paths to analyse, different types of *merging techniques* are used by AE. This article describes these techniques in more detail. The contributions of this article are:

- We present merging techniques used by AE for trading analysis time for flow information precision.
- We present a new technique, *ordered merging*, which may radically reduce AE analysis times for large programs with many possible input variable values.
- We evaluate the effect of the different techniques w.r.t. analysis time and flow information precision on some benchmark programs.

Even though our merging techniques are presented in the context of AE, we believe that they should be applicable for many other WCET analysis techniques where many program paths are explicitly explored.

The rest of the paper is organized as follows: Section 2 presents our WCET tool and AE. Section 3 presents the need for, and the drawbacks of, merging. Section 4 presents related work and other methods which should benefit of our techniques. Section 5 describes our merge point placement techniques and Section 6 presents our new technique for ordering of merge points. Section 7 presents analysis results. In Section 8 we draw some conclusions and discuss future work.

2. SWEET and Abstract Execution

SWEET (SWEdish Execution time Tool) is a prototype WCET analysis tool developed at Mälardalen University [14]. It consists of three main phases; a *flow analysis* where bounds on the number of times different instructions or larger code parts can be executed is derived [9], a *low-level analysis*, where timing cost bounds for different instructions or larger code parts are derived [5], and a *calculation* where the most time-consuming path is found using the information derived in the first two phases [6].

Abstract Execution. AE is a form of symbolic execution based on an AI [4] framework. Rather than using traditional fixed-point iteration, AE executes the program in the abstract domain, with abstract values for the program variables and abstract versions of the operators in the language. For instance, the abstract domain can be the domain of intervals: each numeric variable will then hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. As usual in AI, the abstract value held by a variable, at some point, represents a set containing the actual concrete values that the variable can hold at that point. An *abstract state* is a collection of abstract values for all variables at a point. AE is *input data sensitive*, allowing the user to explore how different input data value constraints affect the program flow [2, 7].

Illustrative example. As an illustration of AE, using intervals as abstract values, please consider Figure 1. When entering the loop, the variable `i` can hold any integer value from 1 to 4. Each execution of an abstract state by the `while` condition might give raise to at least one, and at most two resulting states (the *true* and *false* branch). During the first three executions of the loop condition, there is no value of `i` which terminates the loop. However, the fourth time the condition is executed, `i` will have a value of $[7..10]$, giving that the analysis produces two resulting states. Thus, `i` will have a value of $[7..9]$ at point `p`, and $[10..10]$ at point `r`. Similarly, during the following execution of the

```
i=INPUT; // i=[1..4]
while (i < 10) {
   ...   // p
   i = i + 2;
   ...   // q
}
// r
```

(a) Example

iter	i at p
1	[1..4]
2	[3..6]
3	[5..8]
4	[7..9]
5	[9..9]
6	impossible

(b) Analysis

min. #iter: 3

max. #iter: 5

(c) Result

Figure 1. Example of abstract execution

loop condition both branches can be taken. At the sixth execution of the loop condition, the set of values for the true branch of the loop condition is empty, i.e., only the *false* branch is possible, and AE of the loop terminates.

Deriving flow constraints. The output of AE is a set of *flow facts* [6], i.e., constraints on the program flow, which is given as input to the subsequent calculation phase. To derive these constraints, AE extends abstract states with *recorders*, used to collect flow information. Program parts to be analyzed are extended with *collectors*, which are used to successively accumulate recorded information from the states. For example, in Figure 1 each state may be given a *loop bound recorder* for recording the number of executions it makes in the loop. Similarly, a *loop bound collector* can be used to accumulate the loop body executions (3, 4 and 5 respectively) recorded by the states. The accumulated recordings are used to generate the loop bound constraints in Figure 1(c). Flow constraint generation supported by AE include lower and upper (nested) loop bounds, infeasible nodes and edges, upper node and edge execution bounds, infeasible pairs of nodes, and longer infeasible paths [9].

3. Merging

When using abstract values, conditionals cannot always be decided, as illustrated by the above example. In these cases, AE must then execute both branches separately in two different abstract states. This means that AE may have to handle many abstract states, representing different possible execution paths, concurrently. The number of possible abstract states may grow exponentially with the length of these paths.

In order to curb the growing number of paths, *merging* of abstract states for different paths can take place at certain program points (*merge points*). If the states are merged using the least upper bound operator "$\sqcup$" on the abstract domain of states, then the result is one abstract state safely representing all possible concrete states. Thus, a single-path abstract execution, representing the execution of the different paths, can continue from the merge point. Merge points can be selected at will, but typical placements are after if-statements, and at entries/exits from functions and loops.

Problems with merging. Merging comes with a price, however, since it may yield abstract values that represent concrete values in a less precise way: for instance, the merging of [6..6] and [10..11] yields [6..11], which also contains the concrete values 7, 8, 9 not present in the original intervals. The added values might, if we are unlucky, force AE to execute paths which are not feasible. Figure 2(a) shows an example with an upper loop bound = 4. AE with no merging would find that bound, whereas merging at point p would lead to an overestimated loop bound (8). Merging at point q (at the loop header) instead would lead to a lower overestimation (7) of the loop bound, i.e., also placement of

```
int i, x; // i=[-5..5]
if (i > 0) x=2;
else x=4;
// p
while (x < 10) { // q
   if (isOdd(x)) x++;
   else x=x+2;
}
```

(a) Loop bound overestimation

```
int i;    // i=[-5..5]
int x;
if (i > 0) x=2;   // A
else x=1;         // B
// r
if (i > 1) x=2*x; // C
else x=3*x;       // D
// s
```

(b) Missed infeasible paths

Figure 2. Overestimations due to merging

merge points are important.

Moreover, after a merge we cannot relate the path we executed before the merge with the path we executed after the merge, since several paths might be merged at the merge point. This means that some types of infeasible path calculation cannot be made after merging. Figure 2(b) shows an example where merging at `r` (join after if) leads to that the infeasible paths `A-C` and `B-D` both are missed. This is because the merged state at `r` has a path history where *both* nodes `A` and `B` are included. If we instead postpone the merging to `s` both infeasible paths can be found.

We conclude that merging may lead to a faster AE, but that it also may result in less precise flow information. In Section 5 we describe placement of merge points, as supported by SWEET, allowing us to trade analysis time and flow information precision. In Section 6 we present a new method, based on ordering of merge points, for minimizing the number of concurrent abstract states, and thereby achieving a much faster analysis.

4. Related Work

As mentioned in Section 2, AE can be classified as a combination of AI and symbolic execution [8, 9]. The WCET analysis method by Lundqvist et al. [13], works in a similar fashion, and can potentially also get many parallel states. Compared to Lundqvist's work, AE uses a more detailed value domain, is based on an AI framework, and derives only flow constraints.

In general, our merging techniques should be applicable for any other WCET analysis technique where the paths through the program or parts of the program are explicitly explored. For example, most path-based calculation methods use some type of merging [10, 13, 16, 17]. Similarly, path-enumeration based flow analysis approaches, such as [1, 11], may require merging when the amount of paths grow large.

On an even more general level, both *intra-procedural* and *inter-procedural* program analyses make use of different type of merging and merge points [15].

5. Placement of Merge Points

A crucial question is: where to place merge points? For example, to place them at all join points in the program could mean that we unnecessarily overestimate values, which might lead to less precise flow information, and a non-tight resulting WCET estimate.

```
int complex(int a, int b) {           // BB0
   while (a < 30) {                   // BB1
      while (b < a) {                 // BB2
         if (b > 5)                   // BB3
            b = b * 3;                // BB4
         else
            b = b + 2;                // BB5
         if (b >= 10 && b <= 12)      // BB6 & BB7
            a = a + 10;               // BB8
         else
            a = a + 1;                // BB9
      }                               // BB10
      a = a + 2;                      // BB11
      b = b - 10;
   }
   return 1;                          // BB12
}

int main(void) {                      // BB13
   /* a = [0..18] b = [0..18] */
   int a = 1, b = 1, answer = 0;
   answer = complex(a, b);            // BB14
   return answer;
}
```

Figure 3. Example program

To handle this problem, our method allows the user to control the placement of merge points, in order to explore different tradeoffs between analysis speed and precision. Figure 3 shows an example program (`jcomplex` from the Mälardalen benchmarks [14]), to illustrate the possible placement of merge points. Figure 4 shows the corresponding program CFG where the function call to `complex` has been inlined in `main`, and nodes have been partitioned w.r.t. the functions and loops they belong to. This type of graph (the *scope-graph* [6]) is used by the analysis in SWEET.

The user can currently specify AE to use *no merging*, or one or more of the following merge points: at *function entries* (corresponding to nodes `BB0` and `BB13` in Figure 4), after *function exits* (`BB14`), after *loop body termination*, i.e., at the loop header (`BB1`, `BB2`), after *loop exits* (`BB11`, `BB12`), and at *joins after if-statements* (`BB6`, `BB9`, `BB10`). Note that a node can be of more than one merge point type.

6. Ordered and Unordered Merging

AE and its flow fact generating techniques has been presented in detail in [9]. The original underlying algorithm for processing abstract states is outlined in Figure 5. It is a quite straightforward worklist algorithm, which iterates over a set of abstract states, generating new abstract states from old ones. Abstract states at merge points are moved to a special merge list, and final states are removed. When the worklist is empty, all states in the merge list which are at the same merge point are merged, and the resulting states are inserted in the worklist. The algorithm terminates when both the merge list and the worklist are empty. The algorithm allows any combination of merge point placements (as outlined in Section 5) to be used.

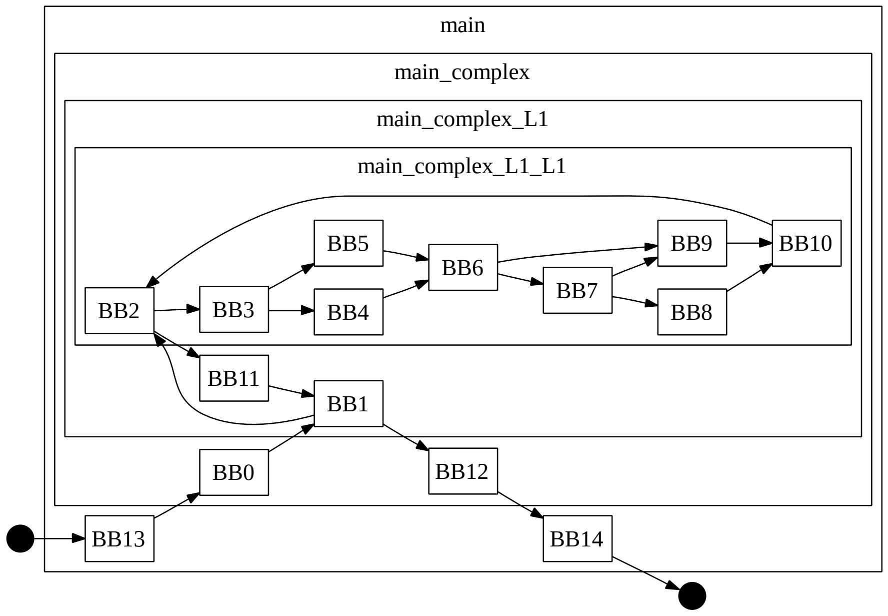


Figure 4. Scope graph for the example program

This algorithm has been successfully used to analyse all the programs in the Mälardalen WCET benchmarks suite [14] during the WCET Challenge 2006 [18]. However, during the industrial case study described in [2], we discovered that for some large programs with many possible input variable values, AE faced complexity problems, i.e., very long analysis times and large amounts of used memory. As an illustration of the inherent problem of the original algorithm, consider the example in Figure 1 again. Further assume that we have decided to use both loop body termination (i.e., `q`) and loop termination (i.e., `r`) as merge points. As explained in Section 2, the fourth time the condition is abstractly executed the analysis will spawn two abstract states, one taking the *true* branch (i=[7..9]), and one taking the *false* branch (i=[10..10]). Both states will eventually reach a merge point (`q` and `r` respectively). However, since they are not in the same merge point they cannot be merged at that time. Instead, both states are moved to the worklist, and will continue their executions in parallel.

We observe that all states resulting from abstractly executing the loop will sooner or later reach `r`. However, there is no mechanism in the original algorithm to force a state in `r` to wait for the other states to reach `r`. Thus, each state that reaches `r` will be continue executing the code following `r` in parallel with the states in the loop. This means that merging in itself is not guarantee to get few parallel states. Moreover, the code after `r` will be executed several times, with almost identical states, which means a lot of unneccessary work.

Ordering of merge nodes. We have designed and implemented an algorithm to solve the problem described above[1]. The basic idea behind the algorithm is to force a state to *wait* for all other states which sooner or later will reach the same merge node at which the state is located. This is achieved by creating an order between all merge nodes and force the processing of states to follow this order. For example, in Figure 1 we want to create an order so states at `r` should wait for all states still executing in the loop. All states resulting from the loop processing can then be merged at `r`, giving that there will be only one state that continues executing after `r`, instead of several.

The algorithm works by first creating an *immediate post-dominance (IPDom) tree* of all nodes in

[1]In the current implementation, merging for recursive programs is not supported.

```
work_list <- {init_state};
merge_list <- empty;
final_list <- empty;
REPEAT
  WHILE work_list /= empty DO {
    s <- select_from(work_list);
    work_list <- work_list \ {s};
    new_states <- ae(s);
    FOREACH s' in new_states DO
      CASE merge_point(s'): merge_list <-
                                 merge_list U {s'}
           final_state(s'): final_states <-
                                 final_states U {s'}
           otherwise: work_list <- work_list U {s'};
  }
  WHILE merge_list /= empty DO {
    s <- select_from(merge_list);
    merge_list <- merge_list \ {s};
    FOREACH s' in merge_list DO
      IF same_merge_point(s,s') THEN
        s <- merge(s,s');
        merge_list <- merge_list \ {s'};
    work_list <- work_list U {s};
  }
UNTIL work_list = empty
```

Figure 5. Original algorithm for AE

the program CFG. Basically, a node n *post-dominates* another node m iff all paths from m to the program exit node intersect n. Similarly, a node n *immediately post-dominates* another node m iff n *post-dominates* m and there is no other post-dominator of m between n and m in the CFG. In the tree, n will be a direct parent of m iff n immediately post-dominates m. In our current implementation, we use the algorithm outlined by Lengauer and Tarjan [12] to create the IPDom tree, with an $O(e\ log\ v)$ time complexity, where e is the number of edges and v is the number of vertices in the CFG.

Secondly, we traverse the IPDom tree bottom-up. For each node traversed, we check if it is a merge node. If so, it is inserted to the end of a *list of merge nodes*. As a result, the list will hold all merge nodes, ordered so that if a node n post-dominates m, then m will be before n in the list. We note that there are often many merge-nodes which do not have any post-dominate relation to one another. Thus, the same set of merge nodes could result in many different orderings, depending on which order the different branches in the IPDom tree are processed. However, for all possible list orderings, it should hold that for all pair of nodes n, m in the list: if n post-dominates m, then m should be before n in the list. The graph traversal has an $O(v)$ complexity, where v is the number of vertices in the CFG.

Thirdly, the list is used to create a *priority queue*, where inserted states are indexed on the position of their corresponding merge node in the ordered list. This queue will replace the merge list in the original algorithm for AE in Figure 5. We also only pop one state at the time from the priority queue into the work list, instead of, as in the original algorithm, popping all states in the merge list to the work list simultaneously. Thus, when extracting a state from the queue, the state with the merge node closest to the beginning of the list will be popped. Moreover, if a state is to be inserted into the priority queue, and there already is a state in the queue at that merge point, we merge the two states and insert the resulting state into the queue (the merged states are deleted). Thus, the number of states stored in

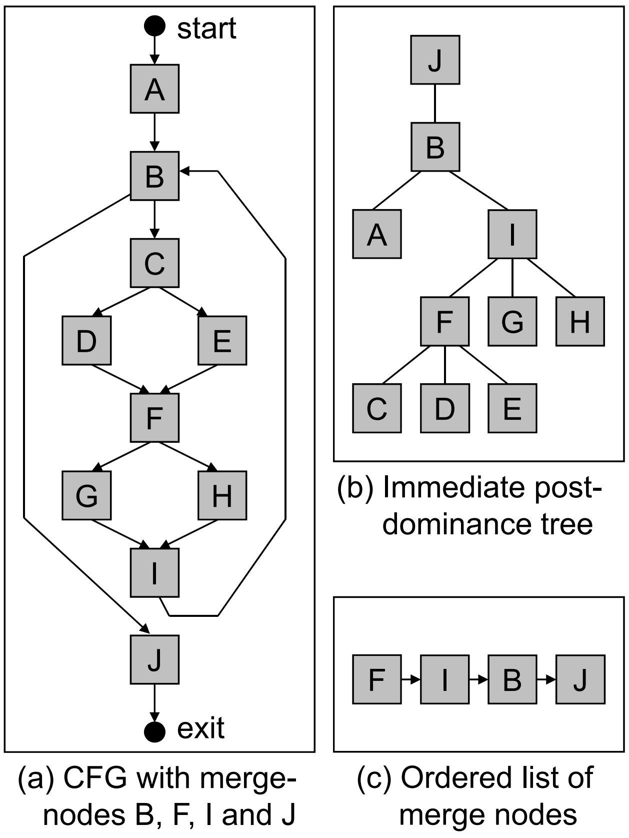

start
A
B
C
D
E
F
G
H
I
J
exit
(a) CFG with merge-nodes B, F, I and J
J
B
A
I
F
G
H
C
D
E
(b) Immediate post-dominance tree
F
I
B
J
(c) Ordered list of merge nodes

Figure 6. Example of merge node ordering

the priority queue will be less or equal to the number of merge nodes in the program.

We have implemented the priority queue as a binary heap, thereby getting an $O(log\ v)$ time complexity for both insertion and popping of states, where v is the number of merge nodes in the CFG [3].

Figure 6 gives an illustration of the ordering of merge nodes. Figure 6(a) shows a CFG with `B`, `F`, `I`, and `J` set as merge nodes, i.e., all possible join points in this program. Figure 6(b) shows the IPDom tree generated for the CFG. Figure 6(c) shows the ordered list of merge nodes resulting from the bottom-up traversal of the tree. The list gives, for example, that a state located in merge point `B` should be processed after states located in `F` or `I`, but before states located in `J`.

7. Evaluation

We have tested our merge strategies on two programs, `jcomplex` and `insertsort`, from the Mälardalen benchmarks using the ARM9 timing model of SWEET. This timing model has not been validated against real hardware. However, we consider it to be a sufficiently realistic "abstract architecture" for the purpose of evaluating flow analysis methods.

For the `jcomplex` program in Figure 3, the input space (i.e., number of possible input combinations) is $19^2 = 361$ values. SWEET's AE analysis time was 7.2 seconds when no merging was used, and the resulting calculated WCET estimate was 918 cycles for the ARM9 target CPU. Table 1 shows the results of the different analyses of `jcomplex`. **ATime** is the analysis time in seconds for AE, and **WCET** is the calculated WCET estimate for ARM9 in cycles. This information is shown for the

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered	**ATime**	8.7	15.0	0.72	0.18	0.84	0.77
merge	**WCET**	918	918	1053	2087	1053	1053
Ordered	**ATime**	8.7	8.3	0.13	0.18	0.18	0.16
merge	**WCET**	918	918	3711	2087	5395	5395

Table 1. Analysis results for `jcomplex`

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered	**ATime**	-	-	1.6	0.08	1.9	1.7
merge	**WCET**	-	-	332	332	332	332
Ordered	**ATime**	-	-	0.09	0.08	0.09	0.09
merge	**WCET**	-	-	332	332	332	332

Table 2. Analysis results for `insertsort`

following five merge point selections: **FE** = at function entries, **FT** = after function exits, **LBT** = after loop bodies, **LT** = after loop exits, **LBI** = after if-statements and loop bodies[2], and **ALL** = after all merge point types.

In the table, we see that merging at function entries and exits (**FE** and **FT**) gives an exact result compared to no merging, however to the cost of long analysis times (even exceeding the no merge time). The other merge selections are efficient, but yield some overestimation. This is mainly due to the conditional updates of the variables `a` and `b` inside the loops, where merging yields overestimation of these variables, which results in an overestimated outer loop bound. We can note that ordered merge gives a larger overestimation than unordered merge. This is probably because the ordered merge gives more merging, with fewer concurrent states, faster analysis but larger overestimation. Ordered merge is the fastest method in all cases.

Table 2 shows the results for `insertsort`, which is a sorting program using the insertion sort method for an array of 10 elements where each element is a positive 32 bit integer (i.e., given a value of $[1..2147483647]$). For this program, the input space is around 10^{93} values. Thus, it is not possible to run the program with all inputs. Nor is it possible to analyse it without merge. However, using the knowledge of the worst case behaviour for insert sort (an inversely sorted array), we can analyse the program using SWEET with the worst case input, and deduce a tight WCET estimate of 332 cycles.

In the table, we see that only some types of merging actually gives a result within a short time ('-' means that the analysis time exceeded 10 minutes). We can see that merging at either loop body termination and loop termination points (**LBT**, **LT**, **LBI**, and **ALL**) gives results in a very short time. This is not surprising, since the number of iterations in the nested loops in the programs is very high, and efficient merge should lower the analysis times considerably. We also see that this efficiency does not give any WCET overestimation penalty for this program. Ordered merge is fastest in all cases.

Table 3 shows the results for `esab_mod`, a larger program provided by one of our industry partners. The program consists of 3064 lines of C code, including 11 functions, 519 conditionals and one loop. The outcome of many of the conditionals are input dependent, and abstract execution of these

[2]We combine these types to ensure that merging always takes place after if-statements in the same iteration.

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered	ATime	-	-	-	-	-	-
merge	WCET	-	-	-	-	-	-
Ordered	ATime	-	-	-	-	163	161
merge	WCET	-	-	-	-	165795	165795

Table 3. Analysis results for `esab_mod`

conditionals may therefore generate many new states. The loop is located in a function with loop bound that is a dependent on the argument to the function. The function itself is called from 372 different call-sites. Since AE does its loop bound analysis fully context sensitive, i.e., we do a seperate analysis for each individual path to the function in the call graph, we get a large number of different calling contexts (8942) for the loop. In the example runs eight variables were given input value ranges, giving an input space of around $1.5 * 10^{12}$ values. Due to this, it was not possible to run the program with all inputs or to manually determine the input value combination that generated the WCET.

For none of the merge point options were we able to finish AE analysis when using unordered merge ('-' means that the analysis time exceeded 1 hour). For ordered merge we were able to finish the analysis within 3 minutes for both the **LBI** and **ALL** options with the same resulting WCET estimate. The ordering of merge nodes took around 82 seconds, and has been included in the total running time for AE. For single value inputs the analysis took around 8-10 seconds (depending on used input values) excluding the time for ordering merge nodes.

All measurements were performed on a 3 GHz PC with 1 Gb RAM, running Linux Ubuntu.

8. Conclusions and Future Work

In this paper, we have described the merge techniques used during abstract execution, a flow analysis method used in SWEET. We have described how to trade the precision of the results for faster analysis, by the use of different placement of merge points and a new merging technique based on sorting merge points. We have shown one example where we can use maximum merging to give a radically shorter analysis time and still get a precise WCET estimate. For another example, however, merging lead to less tight WCET estimates.

Moreover, we have shown that use of merge points is not alone a guarantee to obtain few concurrent states. To handle this, we have presented a new method, based on ordering of merge points, to reduce the number of concurrent states. We have shown that this method is able to significantly shorten the analysis time of large programs with large input spaces.

For future work we plan to investigate the effect of our different merging techniques on different industrial codes, such as the task codes presented in [2]. This will allow us to see, in more detail, how our methods scales for large programs with large input spaces. We will also study the effect of different merge point plecements on different program types and code constructs. This will allow us define guidelines on how to select merging strategy for achieving an optimal combination of analysis time and precision.

References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop of Real-Time Systems*, pages 102–107, June 1996.

[2] Dani Barkah, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Christer Sandberg. Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In *Proc. 20th Euromicro Conference of Real-Time Systems, (ECRTS'08)*, July 2008.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[5] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

[6] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[7] Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering worst-case execution times for software components. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, Pisa, Italy, July 2007.

[8] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

[9] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, December 2006.

[10] C. Healy, R. Arnold, Frank Müller, David Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[11] D. Kebbal. Automatic flow analysis using symbolic execution and path enumeration. In *Proc. of the 2006 International Conference Workshops on Parallel Processing (ICPPW'06)*, pages 397–404, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, pages 121–141, July 1979.

[13] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.

[14] Mälardalen University. WCET project homepage, 2007. `www.mrtc.mdh.se/projects/wcet`.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2nd edition*. Springer, 2005. ISBN 3-540-65410-0.

[16] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[17] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01)*, November 2001.

[18] L. Tan. The worst case execution time tool challenge 2006: The external test. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

ON COMPOSABLE SYSTEM TIMING, TASK TIMING, AND WCET ANALYSIS[1]

Peter Puschner[2] and Martin Schoeberl[3]

Abstract
The complexity of hardware and software architectures used in today's embedded systems make a hierarchical, composable timing analysis impossible. This paper describes the source of this complexity in terms of mechanisms and side effects that determine variations in the timing of single tasks and entire applications. Based on these observations, the paper proposes strategies to reduce the complexity. It shows the positive effects of these strategies on the timing of tasks and on WCET analysis.

1. Introduction

Until the early 90s of the 20th century the computer architectures used in embedded hard real-time systems were relatively simple. In the light of these simple architectures, hierarchical (composable) timing models – models that separated the low-level task timing issues from the real-time scheduling problem at the high level – had been conceived. Worst-case execution-time analysis (WCET analysis) had started to become an independent field of investigation within real-time systems research.

Since then, researchers working on WCET analysis have developed methods to identify (in)feasible paths through pieces of code and strategies to compute WCET estimates for code running on different hardware architectures. Over the years, the results of WCET analysis allowed its researchers to compute WCET estimates for code running on more and more complex computer architectures.

While new computer hardware had entered the stage and advances in WCET analysis were made to deal with these changes (e.g., to model the effects of instruction pipelines and caches on task timing), the overall timing models still remained unchanged. I.e., although the temporal (de)composability of task execution times had been lost, real-time schedulers and schedulability analysis still use the strategies that had been conceived at a time when task execution times were independent. The only available measure to deal with the lack of composability in task timing is the addition of an extra analysis step. This analysis uses information including the periods, priorities, and the physical-memory maps of tasks to make a pessimistic assessment of the worst-case effects of the timing interactions between tasks, see, e.g., [14].

Tools that assess the worst-case side effects of task executions on overall system timing are valuable at the moment. In the long run, however, we will have to get rid of the side effects of tasks on system timing instead of analyzing these effects. Only the elimination of the side effects provides the basis for a development and analysis process that is hierarchical and thus much less complex than what is currently state of the art. It is therefore the purpose of this paper to identify the properties of current

[1]The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 214373 (ARTISTDesign).
[2]Vienna University of Technology, A1040 Vienna, Austria; email: peter@vmars.tuwien.ac.at
[3]Vienna University of Technology, A1040 Vienna, Austria; email: mschoebe@mail.tuwien.ac.at

task-execution models that adversely affect the decomposability of tasks and make timing analysis difficult. Based on these findings we introduce a restrictive hardware and software architecture that allows us to return to a hierarchical timing analysis of manageable complexity. The discussion will highlight the consequences of this proposal on task design, on task timing characteristics, and – as a consequence – on WCET analysis.

2. Hierarchic Timing Analysis

Traditionally the timing analysis of hard real-time systems, comprising the CPU-time scheduling or schedulability analysis and the worst-case execution-time analysis, is a hierarchical process. At the lower level, tasks are analyzed for their worst-case execution time (WCET). The results of this low-level analysis are used either for constructing a CPU schedule that meets the timing constraints of the application or for the purpose of schedulability analysis – the latter ensures that a task set characterized by its task parameters (including execution times) will indeed be scheduled correctly.

A hierarchic decomposition as sketched above reduces the complexity of the timing analysis and the real-time systems engineering process. According to [13], such a decomposition requires that the subsystems are nearly decomposable, i.e., *the interactions among the subsystems are weak but not negligible* [13].

2.1. A Model of Simple Tasks

In this work, our main interest is to investigate phenomena related to the time consumption of tasks. For this purpose we assume that all tasks are *S-tasks* [6], simple tasks that do not have any synchronization points inside. As a precondition to the execution of each task instance we assume that all inputs for the task are available. During task execution there is no I/O or other blocking. A simple task produces outputs by writing to defined locations within its local memory – the availability of the outputs is part of the postcondition of each task execution. Outputs are read and further processed (copied or transmitted in a message) by the operating system after the completion of the task.

For the sake of simplicity we assume that tasks are *stateless*, i.e., they do not preserve any data values between invocations. Note that tasks that need a state can be converted to stateless tasks by making their state an input/output variable. Each instance of such a converted task reads this input/output variable after its start and writes its contents back as an output before it terminates. So the next instance can read in the “state” again, and so on.

The interface between a task and its environment (the other tasks and the physical world) is characterized by its control properties, temporal constraints, the functional intent and its data properties [6]. Tasks communicate with the environment with the help of the operating system, that reads resp. writes the interface (input/output) variables of the tasks. One can observe that the interactions between the tasks of a computer system influence the execution times of the tasks. Some interactions (e.g., the exchange of data) are due to phenomena that are observable parts of the task interface. Other interactions (e.g., the modification of the contents of a shared cache which, in turn, influences the execution time of other tasks) cannot be traced back to the task interface. The latter are called side effects.

During the execution of a task instance, the values of the interface variables of the task are not defined. Therefore it is not safe to access these variables from outside the task while the task is in progress. The termination of the task commits the outputs. The outputs can then be propagated and processed by the task environment.

2.2. Simple and Complex Computer Architectures

In *simple computer architectures* [6], the timing of each instruction can be determined locally, from the knowledge of the instruction, the operands of the instruction, and a small execution context of the instruction (often consisting only of the instruction itself). In these architectures, the execution of an instruction can influence the execution time of another instruction in another task only via the explicit change of values in a data memory that it shares with one or more other tasks. There are, however, no implicit effects (side effects) of task behaviour on the execution times of other tasks. This way we have clearly defined interactions between subsystems (tasks) that can be taken into account by the scheduler, resp. schedulability analysis at the system level. Timing interactions between tasks are limited to the effects of the data exchanged at the interfaces of the tasks.

In *complex computer architectures* the interactions between the different subsystems (tasks) are in general not weak. This is because interactions are no longer restricted to a limited amount of explicit timing dependencies that are caused by data sharing (including access to physically shared memory as well as message communication) via the interfaces of the tasks. These architectures are characterized by a number of additional timing effects due to the competition in the reservation of and access to scarce, shared system resources. Depending on the particular computer system architecture shared resources include pipelines for processing instructions and loading data, instruction and data cache memories, and fast on-processor caches or registers for speculative branch and trace prediction. Besides these mechanisms, the sharing of buses and memories for instructions and data among the processors of chip-multiprocessing systems adds another source for temporal side effects between tasks.

Because complex computer architectures are more and more used in embedded real-time systems, we have to be aware of these implicit side effects that influence execution times. We have to identify and analyze these interactions, and we have to investigate into appropriate ways of eliminating the hidden side effects, thus avoiding that the reasoning about the temporal properties of real-time systems becomes unmanageably complex.

3. Interactions of Task Timing

In the following we investigate the timing interactions between tasks in more detail. We describe both the reasons for the interactions and the resulting phenomena that can be observed.

3.1. Task Interactions in Simple Architectures

In general the execution times of simple tasks vary. Such variations are due to effects of differences in the inputs that are passed to the task via its interface.

Variable, data-dependent execution times of CPU instructions: A number of processors implement (part of) the CPU instructions in micro code where more complex instructions execute repetitive steps over a number of CPU clock cycles. In some cases the number of steps depends on the actual operands which leads to execution-time variations (e.g., shift, multiply, and division).

Consequences for task timing analysis: If the operands of instructions with data-dependent timing are not exactly known at analysis time, static analysis has to use pessimistic abstractions instead. In measurement-based analyses, variable instruction timing in general increases the number of different execution-time combinations of instructions that have to be compared in search for the worst case. Roughly, every instruction with k different execution times increases the number of different scenarios by a factor k. To deal with this increase in complexity, one either needs to use knowledge about the instruction behaviour when generating input data for measurements or, alternatively, increase the number of test cases significantly to obtain results of sufficient quality.

Different execution paths: In numerous algorithms the conditions of conditional branches that determine the actual instructions executed during an execution directly or indirectly depend on the task inputs. Upon execution, differences in inputs result in different test results in conditionals which, in turn, produce different execution traces or paths, and thus possibly different execution times.

Consequences for task timing analysis: As every conditional branch essentially doubles the number of possible execution paths – an effect that multiply occurs in loops – the number of possible execution paths of a task is usually too high to analyze the timing of all possible paths one by one. As a consequence, static timing analysis uses pessimistic abstractions which, in turn, can lead to overestimations in the result of WCET analysis.

3.2. Task Interactions in Complex Architectures

In addition to the above-mentioned explicit timing interactions between tasks, the following mechanisms of complex computer architectures give rise to temporal side effects.

Intra-task effects on hardware state: Let us at this point consider a single periodic task that is perfectly shielded from external side effects. Still, the different instances of this task, operating on different inputs, may execute on different paths. This leaves the hardware of the computer system (e.g., the instruction cache, branch prediction buffers, etc.) in different states when the task completes. As the hardware state at termination equals the start state of the next instance of the tasks, each execution of a task influences the timing of subsequent instances of the task. Depending on whether the hardware state evolves monotonously and converges to a fixed point after a certain number of executions or not (as in the case of conditional cache conflicts within single task instances), the state effects on execution times may stabilize or not.

Effects on task timing: One observes variations in execution time due to different hardware states at the task start. In particular the first instance of a task cannot benefit from state changes (e.g., the loading of cache lines) of previous executions, thus usually consuming much more time than follow-up instances. Depending on whether there are conflicting state effects within task instances, timing effects between instances may disappear after the state has reached its fixed point or not. Even if a such a fixed point exists, the number of executions needed to reach the fixed point is in general unknown.

Consequences for task timing analysis: Which starting state should be assumed? What is the "worst-case starting state", i.e., the starting state from which an execution of maximum duration starts? Do we really want to consider the worst-case starting state in the execution time analysis given that the cost for building up the state in the first instance of a task is usually much higher than the state-dependent cost of successive executions, or shall we discern between the first/first N – what N? – and all other executions that follow?

In static WCET analysis, the pessimistic approximation of possible start states leads to pessimism in the computed WCET bounds. In measurement-based analyses, intra-task state effects increase the space of parameters that are relevant for execution times, which has a negative effect on the percentage of possible situations that can be assessed with a given set of resources.

External hardware state modifications **between invocations** ***(no preemption):*** So far we assumed that the hardware state of a task is not modified between successive executions. In real-world scenarios, however, other tasks and operating system activities alter the state left by the task, and thus the starting state of further instances.

Effects on task timing: The effects on timing are not limited to the intra-task effects mentioned above. Although state changes from prior executions of the task may still be effective, other tasks and operating system activities interfere with the task state, i.e., there are task-external influences on the task execution times as well.

Consequences for timing analysis: As above, there is the question about which start states are relevant for WCET analysis, resp. which abstractions should be used for the analysis. In contrast to the previous discussion, timing effects are not local to the task, however. Therefore, besides the WCET analysis one needs some extra, global analysis to account for task interferences between task executions and their effects on the overall timing of the real-time computer system. A question in this context is: What are useful abstractions for each of these analysis steps in order to achieve a clean separation between WCET analysis and the timing analysis that accounts for the interferences?

External modification of state **during execution** ***(preemption):*** In systems with preemptive scheduling, the situation gets even more complex as preemptions may have almost arbitrary effects on a the state of a task during its execution.

Effects on task timing: The effects of preemptions on the state of a task depend on a number of factors, e.g., the number of preemptions, the state of the task at preemption time, the state modifications performed by the preempting code.

Consequences for timing analysis: As above, in addition possible interferences during task preemptions have to be considered, which again adds complexity/pessimism to the analysis. A simple hierarchical timing analysis that decomposes into a low level WCET analysis and a high-level scheduling or schedulability analysis is beyond reach because of the strong interactions between the two levels.

Dynamic state-sensitive resource allocation and scheduling: Actual out-of-order processors perform speculative execution even over predicted branches where the branch outcome is not yet known. As a consequence around 100 instructions[2] can be in the pipeline on the fly between instruction fetch and instruction retirement.

Effects on task timing: The execution time of a single instruction depends on a very large execution history. Assuming a flushed pipeline on a basic block start is not an option anymore.

Consequences for timing analysis: Modeling the state of about 100 instructions per clock cycle and the speculative execution will result in a state space explosion. The situation can get worse in the

[2]On a Pentium 4 the minimum latency of an instruction between fetch and retire is 31 clock cycles and up to 3 instructions can be fetched each cycle [1]. Register renaming restricts the number of micro operation in execution to 128.

presence of so-called timing anomalies, i.e., when the dynamic scheduling of instructions can lead to non-monotonic timing relationships between instruction sequences and the constituents (parts) of these sequences. The latter poses a major obstacle to a safe compositional timing analysis.

3.3. Task Interactions in Chip-Multiprocessors

Due to the ever increasing transistor budget [8] for a chip several functions (or IP cores) can now be integrated into a single chip. This integration is often referred to as System-on-a-Chip (SoC). SoC is categorized into two types:

Heterogenous multiprocessors contain a number of different IP cores (e.g., general purpose processor, DSP co-processors, small memory units) on a single chip. The cores are connected either by point-to-point links or by a network on chip (NoC). Those systems are called multi-processor SoC (MPSoC).

Homogenous multiprocessors contain several identical processors with some on-chip memory. Those systems are also referred as chip multiprocessors (CMP).

Both architectures are common in embedded systems. Due to the power wall [1] CMP systems are now also state-of-the-art in desktop and server processors. In this paper we consider CMP systems and the impact of the shared memory on the timing analysis. Three, quite different CMP architectures are state-of-the-art: (1) multicore versions of super-scalar architectures (Intel/AMD), (2) multicore chips with simple RISC processors (Sun Niagara), and (3) the CELL architecture.

Most cores for CMP allow fine-grain multithreading within a single core. Multithreading in a core can hide latencies due to cache misses. With simultaneous multithreading (SMT) more than one thread can execute in a single pipeline stage when enough functional units are available. Multithreading increases throughput for server type workloads due to higher processing resource utilization; the individual task execution time increases.

Complex processor CMP: Mainstream desktop processors from Intel and AMD include two or four out-of-order executing processors. Those processors are just replications of the original, complex cores that share a 2nd level cache and the memory bus. Cache coherence protocols on the chip keep the level 1 caches coherent and consistent. Furthermore, those cores also support SMT, sometimes also called hyper-threading.

RISC based CMP: Sun took a completely different approach with their Niagara T1 [5] by abandoning the super-scalar architecture that tries to extract instruction level parallelism (ILP) from sequential code. Eight simple cores implement fine-grain multithreading to support thread level parallelism often found in server workloads. Each core consists of a simple six-stage, single-issue pipeline similar to the original five-stage RISC pipeline. The additional pipeline stage adds fine-grained multithreading. Four threads are supported on each core that are scheduled in round-robin fashion. With 8 cores the Niagara can execute 32 independent threads of execution. When a thread stalls due to a cache miss or a load-use dependency it is skipped in the schedule. The first version of the chip contains just a single FPU that is shared by all 8 processors.

Local memory based CMP: The Cell multiprocessor [2, 3, 4] takes an approach similar to a distributed memory multiprocessor. The Cell contains, beside a PowerPC microprocessor, 8 synergistic processors (SP). The SPs contain 256 KB on-chip memory instead of a cache. The PowerPC, the 8 SP, and the memory interface are connected via a 4 ring network. Communication between the cores in the network has to be setup explicitly. All memory management, e.g. transfer between SPs or between on-chip memory and main memory, is under program control, resulting in a new programming model.

Simultaneous multithreading: The tight coupling of the CMP cores introduces several timing interactions that are hard to predict. The simplest form of multiprocessing within a single pipeline is introduced by fine-grain or simultaneous multithreading. The hardware managed threads of execution interact in a very fine grain manner: each stall in one thread influences the execution time of the other threads. The best WCET estimates we can provide for hardware multithreading is the same time as executing those threads serially on the same pipeline.

Keeping caches coherent and consistent: Cache coherence protocols (bus snooping or directory based) enforce a coherent and consistent view of the main memory. These protocols exchange the cache information between all cores on each memory access and introduce a high variability of the cache access time even when the access is a cache hit.

Shared caches and memory: Probably the main source of timing interaction comes from the shared 2nd (and probably 3rd) level of cache and the shared main memory. The shared memory provides an easy-to-use programming model at the cost of unpredictable access time to the data. With global multiprocessor scheduling a task can migrate from one core to another – even within a single period of execution. A migrated tasks completely looses its L1 cache state.

4. Avoiding Unwanted Interactions

We have seen that a number of factors contribute to variations in task execution times. Some of the effects are malign as they are not local to a single task execution but invalidate the hardware state that other tasks or other instances of the same task have built up. These interactions cause side effects that obstruct a hierarchical timing analysis.

In this section we propose some ways to eliminate these interactions. The central idea is to protect the time-relevant state of a task from dynamic changes that make it unpredictable. To this end, we aim at the spatial separation of tasks and we replace dynamic run-time decisions by unalterable, pre-planned control mechanisms where all decisions have been taken offline, at implementation time. In detail, our solution builds on the following mechanisms:

- The use of single-path code in all tasks,
- The execution of a single task/thread per core,
- The use of simple in-order pipelines, and
- Statically scheduled access to shared memory in CMPs.

4.1. Use of Single-Path Code

When considering simple architectures, we think that data-dependent instruction execution times can be eliminated easily. In fact there are a number of processors with constant instruction execution times around. Timing variations due to the execution of different execution paths and the intra-task effects on the hardware state that occur in complex architectures can be eliminated by transforming the code into so-called single-path code [11]. In the single-path transformation [9], all control dependencies in the code are removed. Instead, the input-dependent conditionals are replaced by predicated instructions that have invariable execution times.

Effects on task timing: As single-path code always executes the same trace, there are no execution-time variations due to multiple paths in simple architectures. For single-path task implementations, execution-time variations due to intra-task timing effects are restricted to the warm-up phase of the task. As all executions of a task run the same trace the hardware state stabilizes after a limited number of executions. After this fixed point has been reached, the timing of the task remains constant. To avoid that the single-path conversion yields code with very poor performance, we suggest the use of *WCET-oriented* programming strategies and algorithms [10].

Consequences for timing analysis: The timing analysis of single-path tasks is trivial. On simple architectures it is sufficient to measure the execution of a single task instance, with any input data, to obtain the (single) execution time of the task. For the complex architectures, the execution time of isolated tasks can be measured after a limited number of executions, once the hardware state has stabilized at its fixed point.

4.2. Execution of a Single Task per Core

Both types of external modifications (inter-task effects) of the hardware state of a task – those occurring between invocations and those due to preemptions – can only be eliminated by protecting the hardware state against influences from other tasks. One way to achieve this is saving and restoring the state whenever a task completes or a task gets preempted. As the administrative overhead for this state management seems to be pretty high, we propose a more rigorous shielding of tasks that benefits from the current trends in hardware development – assigning each task/thread to a dedicated core of a chip multiprocessor. As the used simple tasks do not access shared data during their execution, each processor builds up its own private state and a spatial separation of the timing relevant state of the tasks is achieved. The timing impact of accesses to shared data for the purpose of communication and I/O (as performed by the operating system) does, of course, need special consideration. The latter will be discussed below (see Section 4.4).

Effects on task timing and timing analysis: Assigning each task to a dedicated processor core eliminates all of the mentioned inter-task timing effects. This way, task timing analysis only has to consider task-internal effects on the state. This, in turn, can reduce the state space of the execution-time anaysis significantly, thus yielding tighter (static analysis) or safer (measurement-based analysis) results of WCET analysis. In addition to simplifying the task timing analysis, the elimination of task interactions reduces the side effects on task timing, thus allowing for a better composibility in the overall timing-analysis process.

4.3. Simple in-order CMP Pipelines

Extracting ILP from sequential code in one task with speculating out-of-order pipelines consume a lot of resources and is hard to analyze. For time predictable systems the transistor budget for future CMPs is better spent by replication of simple RISC pipelines. The additional available cores can be utilized to shield individual tasks as proposed in the former section. We assume local data and instruction memory per core either program managed or organized as a cache.

Current trends in computer architecture actually simplify WCET analysis. Besides the Intel/AMD approach, the CMP pipelines are simpler than the former mainstream complex processors. The architectures target at thread-level parallelism instead of ILP – an abstraction that can be better handled in real-time systems.

The pipeline of Sun's Niagara CMP is a simple in-order pipeline where timing anomalies [7] are not an issue anymore. The CELL SP elements are dual-issue SIMD in-order pipelines. The SP contains no cache and no virtual memory. In the CELL processor each data exchange between the cores has to be setup under program control. Therefore, we can apply the time triggered approach of data exchange on the CELL architecture.

Effects on task timing and timing analysis: Single issue, in-order execution pipelines are well understood for WCET analysis. The speedup due to pipelining can be modeled for basic blocks and also for larger constructs, such as loops.

4.4. Statically Scheduled Access to Shared Memory

In a CMP system the competition for shared resources shifts from the CPU to the memory bandwidth. Imagine an extreme CMP system with more CPUs than tasks to execute. In that case there is no competition for the CPU – we even can avoid scheduling at all. However, all CPUs access the single global memory. A shared resource where the access has to be scheduled, e.g., through an arbiter. Even when this example is not practical at the moment, it shows the trend towards integrating the competition for the memory bus into the timing analysis.

We consider a static, preplanned scheduling of the memory access [12] for all cores. The arbitration of the memory access is time sliced. Integrating the knowledge of the access time slices into the WCET analysis provides safe estimates for load/store instructions and instruction cache fills. The time slicing does not have to be regular. We can introduce a relative boost (longer slices) for some cores at the cost of other cores that run tasks with enough slack time.

To avoid hard-to-predict task-migration (from one core to another) costs we pin each task to a dedicated core. If a CPU supports hardware multithreading, only one of the virtual CPUs can be used. The other virtual CPUs need to be disabled.

Consequences for timing and schedulability analysis: The scheduling of the memory access has to be integrated into the WCET analysis. With a static schedule of the memory arbitration the access time property is well known and independent from the activity of the other cores. With few tasks – or even a single task – executing on a core, the traditional scheduling for the CPU resource disappears. Scheduling is performed at the memory access level. The low overhead of a task switch at the memory arbitration allows fine grain access control: either time sliced down to a single memory access or a percentage based bandwidth scheduling are feasible.

5. Summary and Conclusion

In this paper we investigated into the problems of nowadays timing analysis. We showed that, due to the properties of the used hardware and software architectures, tasks cannot be considered independent in their execution. As a consequence, task timing is not an isolated property, which makes the analysis of both task timing and system/application timing highly complex.

We analyzed the reasons for the complexity of timing analysis and identified ways to make a hierarchical compositional timing analysis possible. Our solutions utilizes the architectural features of chip multiprocessors that bring along performance and the parallelism we need to reduce the resource competition between tasks.

- For each *single task* we make task timing easier to predict and stable, the latter meaning that each execution of a task has the same execution time. Regarding software, we use the single-path conversion to reduce the number of execution paths (or traces) to one, and WCET-oriented programming to get reasonable performance. On the hardware side we use processors with constant instruction execution times. Further, we rely on in-order pipelines to eliminate the effects of dynamic instruction scheduling, a central source of timing anomalies.
- When considering the *whole task set* of an application, our main goal was to eliminate the inter-task timing effects. By allocating each task to a dedicated CPU core we avoid those timing interferences that are due to the competition for scarce CPU and memory resources. The pre-runtime, offline planning of all accesses to shared memory removes all other interferences, which are due to the – necessary and inevitable – data exchange between tasks.

To summarize, the introduced mechanisms simplify the structure of single tasks and shield different concurrent tasks from one another, both in the spatial and in the temporal domain. The mechanisms both simplify the overall timing analysis – in that they make a hierarchical timing analysis possible – and in parallel simplify the execution characteristics of tasks, thus paving the way back to a simple WCET analysis.

References

[1] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[2] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262, 2005.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *j-IBM-JRD*, 49(4/5):589–604, 2005.

[4] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006.

[5] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multi-threaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[6] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[7] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[9] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[10] Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[11] Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[12] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*, pages 49–60, Dec. 2007.

[13] Herbert Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, 3rd edition, 1996.

[14] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. 17th Euromicro Conference on Real-Time Systems*, pages 41–48, Jul. 2005.

TOWARDS A COMMON WCET ANNOTATION LANGUAGE: ESSENTIAL INGREDIENTS[1]

Raimund Kirner, Albrecht Kadlec and Peter Puschner[2]
Adrian Prantl, Markus Schordan and Jens Knoop[3]

Abstract
Within the last years, ambitions towards the definition of common interfaces and the development of open frameworks have increased the efficiency of research on WCET analysis. The Annotation Language Challenge *for WCET analysis has been proposed in line with these ambitions in order to push the development of common interfaces also to the level of annotation languages, which are crucial for the power of WCET analysis tools.*

In this paper we present a list of essential ingredients for a common WCET annotation language. The selected ingredients comprise a number of features available in different WCET analysis tools and add several new concepts we consider important. The annotation concepts are described in an abstract format that can be instantiated at different representation levels.

Keywords: Worst-case execution time (WCET) analysis, annotation languages, WCET annotation language challenge.

1. Why a Common WCET Annotation Language?

The situation for WCET analysis is very heterogeneous. Within the real-time community it is a well known fact that manual annotations are needed to assist non-perfect analyses. Various tools exist providing different levels of sophistication [19]. However, as the *WCET Tool Challenge* [6] has shown, few tools share the same target hardware, analysis method or annotation language.

While a multitude of targets is beneficial and a diversity in tools and methods is favorable, a common annotation language is *required* for an accepted set of benchmarks in order to evaluate the various tools and methods. Still, as a direct consequence of the first WCET Tool Challenge a set of accepted benchmarks has already been collected, without such annotation support.

To enable common annotations within these benchmarks, the *WCET Annotation Language Challenge* [12] has formulated the need for a common annotation language. This language is a means of specify-

[1]This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract No P18925-N13, by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

[2]Institut für Technische Informatik, Vienna University of Technology, Treitlstraße 3/E182.1, Wien, Austria, email: {raimund, albrecht, peter}@vmars.tuwien.ac.at

[3]Institut für Computersprachen, Vienna University of Technology, Argentinierstraße 8/E185.1, Wien, Austria, email: {adrian, markus, knoop}@complang.tuwien.ac.at

ing the *problem-inherent information* in a tool- and methodology-*independent* way, supporting, e.g., static analysis equally well as measurement-based methods, thus allowing the combination of their results. It must also be expressive enough to master the difficult task of providing annotations at the *source* level, which is the natural specification level, as well as supporting the annotation of binary or object code, if the source code is not available, e.g., for closed sources like operating systems or libraries.

Therefore, a common language may allow the tool developers to concentrate on their analysis methods, creating interchangeable building blocks within the timing analysis framework, as intended by ARTIST2 [10]. By using this common annotation format as a common interface, tools can evaluate the same set of sources for a fair comparison of performance and may exchange analysis results to synergetically supplement each other. The steps of manual annotation, automatic annotation and timing analysis can be repeated, thus iteratively refining the analysis results.

All this should foster common established practices and may, eventually, lead to standardization, resulting in a broader dissemination of WCET analysis throughout research and industry.

2. Basic Concepts

2.1. Definitions

Flow Constraints: We define *flow constraints* to be any information about the control or data flow of a program code. Data flow, however, is not only meant in the sense of *def-use chains*, but, for example, variable-value ranges at program locations. Typical examples of flow constraints are loop bounds or descriptions of (in)feasible paths.

Timing Constraints: We define *timing constraints* to be any information that is introduced in order to describe the search space of the WCET analysis. Because control and data flow represent the basis for the WCET analysis, the *flow constraints* of a program are always part of the *timing constraints*. An example of a *timing constraint* not being a *flow constraint* is the specification of access times of different memory areas.

Constraints versus Annotations: We distinguish between the *timing constraints* and the *timing annotation* of program code. The timing constraints are the information per se and the timing annotation is the linkage of the timing constraints with the program code.

There are different possibilities of how to annotate the program code with timing constraints. One possibility to annotate the program is to write the timing constraints directly into the source code, either as native statements of the programming language or as special comments. It is also possible to place timing constraints in a separate file, if the source code may not be changed.

If a programmer has to annotate the program modules at different representation levels a common syntax for the different representation levels would be especially beneficial and useful.

2.2. Layers

The WCET of a program cannot be determined precisely without knowing information about the target-computer platform on which the program will be used. The computer platform of a program includes, for example, the development tools, the operating system, the hardware, and the application environment. Naturally, the computer platform is sliced into layers to benefit from the independence of different parts that constitute the computer platform. For example, the operating system is an optional layer that may be placed on top of the hardware layer, and again, the layer of the development tool chain may be on top of the operating system.

These layers are the key to the *reuse* of timing annotations in case a layer is changed. For example, if we change the processor type (hardware layer) but still use exactly the same code binary, any timing constraints describing the behavior of the *build-and-run layer* can still be reused, if it does not specify explicit times.

A prerequisite for the smooth replacement of layers is that each annotation has a layer specified in its definition. A layer is replaced by disabling the current instance of the layer and enabling another one as input for the analysis.

Note that the layers are not fixed, but rather open for extensions. For example, if an operating system delivered in binary form has different absolute times specified for different processor types, it does make sense to specify them in a combined OS/HW layer besides the other OS and HW layers.

2.3. Validity of Timing Constraints (Timing Invariants versus Fictions)

The goal of WCET analysis is to calculate a precise WCET bound. However, the developer might also be interested in experimenting with the timing constraints to analyze changes of the program behavior, e.g., to tune the system. For example, the developer might specify a fictive loop bound to determine the influence of the loop on the overall timing. As another example, the developer might want to test an absolute time bound for a code section independently of the real execution time. In both scenarios, timing constraints are not necessarily used to describe a superset of the real program behavior.

In WCET analysis research, program annotations are typically assumed to describe a superset of the possible system behavior, i.e., system invariants. We extend this annotation concept to information that does not have to be a superset of the system behavior. We call all timing constraints that describe a superset of the possible system behavior *timing invariants*. In contrast, we introduce *timing fictions* as arbitrary timing constraints the user might want to use for experimenting with the timing behavior of the system. We add a flag to each timing annotation to mark it either as a *timing invariant* or a *timing fiction*.

The intention of introducing *timing fictions* is not to foster its use for WCET analysis, because *timing fictions* may cause an underestimation of the WCET. But in case that a developer wants to experiment with the sensitivity of the timing behavior, then it is an additional safety feature if the user is able to explicitly mark such timing constraints as *timing fictions* and has to enable them explicitly to be included in the analysis.

Definition 2.1 *(Timing Invariant): A timing constraint C is a* timing invariant *at its associated annotation layer L, iff for all possible systems that use annotation layer L, it holds that for all possible initial system states the system execution fulfills the timing constraint C. If a timing constraint is associated with more than one layer, then the condition has to hold for all possible systems that use all of its associated layers.*

Definition 2.2 *(Timing Fiction): If a timing constraint C is not a* timing invariant *at its associated annotation layer, then it is a* timing fiction.

In the case that *timing invariants* and *timing fictions* are in conflict, the semantics of *timing fictions* is to override conflicting *timing invariants*. Whenever a *timing invariant* is overridden due to a *timing fiction*, the WCET analysis tool should give a log entry to the user.

The following provides examples of *timing invariants* and *timing fictions*:

```
void f (int a, char[] b)
{
  int i;
  a = a % 20;
  for (i=0; i<a; i++) //loop1
  {
    if (i%2 == 0)
      b[i] = a; //m1
    else
      b[i] = 0; //m2
  }
}
```

Timing Invariant:
Expressing as linear flow constraint that the `then`-path is executed at least as often as the `else`-path: $m1 \geq m2$ (see annotation **C2.3**)

Timing Fiction:
Specifying a lower and upper loop bound of 40: $LB(loop1) = 40 \ldots 40$ (see annotation **C2.1**)

In the *timing fiction* example with loop bound $LB(loop1) = 40 \ldots 40$, an IPET-based WCET analysis tool typically transforms the program structure into flow equations and the fictive loop bound is transformed into a flow constraint. In this case, the *timing fiction* redefines the execution count of control-flow edges in the final WCET calculation.

2.4. Checking of Invariants

Manual annotations are potentially error-prone and may yield incorrect WCET estimates. In the case that timing constraints originate from the operation environment it is, however, possible to "lift" operation environment information to the *program layer*, e.g., by inserting range checks and similar assertions wherever appropriate.

```
int count = read_from_sensor();
while (count ≥ 0) {
  count--;
  ...
```

If we assume that the environment dictates that the return value of read_from_sensor() is in the interval [0,47], an upper loop bound of 48 would be an *invariant at the operation layer* and a *fiction at the program layer*.

```
int count = read_from_sensor();
assert(count < 48);
while (count ≥ 0) {
  count--;
  ...
```

However, if we specialize the program by inserting an assertion, the loop bound of 48 becomes an invariant at the program layer.

As a result of lifting annotations to the program layer, the resulting program becomes a specialized instance of the original program. Because the assertions allow the compiler to perform additional optimizations, the specialized program can also have better performance than the original program. These kinds of assertions can easily be generated by an automatic tool and could be valuable for diagnosis and testing of annotations. An example of using runtime checks with special support by the compiler is Modula/R: the Modula/R compiler optionally generates for each source-code location that is referenced by a timing constraint a separate counter variable where an exception is raised at runtime if their specified bound is exceeded [17].

3. Ingredients of the WCET Annotation Language

In the following we describe essential ingredients for a WCET annotation language. The different timing constraints are described at a conceptual level without focusing on the concrete syntax of an annotation language. We use ANSI C code examples to illustrate the usefulness of the different timing constraints. The definition of a concrete syntax is beyond the scope of this paper. We propose the following categories of ingredients, which are detailed in the rest of this section:

C1 Annotation Categorization
C2 Program-Specific Annotations
C3 Addressable Units
C4 Control Flow Information
C5 Hardware-Specific Annotations

C1 Annotation Categorization

We define attributes for timing constraints to categorize and group them. These categorization attributes help to organize, check, and maintain timing annotations. Supporting the maintenance of timing annotations is a very important aspect to improve the correctness of timing constraints. For example, if a user writes an annotation with speculative constraints just for testing the influence on the timing behavior of the system, there is the potential danger that he/she forgets to remove such an annotation from the program later on. Further, whenever code is reused or parts of the computer platform are changed, it is necessary to identify those annotations that have to be checked or adapted. The categorizations **C1.1**, **C1.2**, and **C1.3** are orthogonal categorizations, but their joint use is intended.

C1.1 Annotation Layer

Each timing constraint has associated an *annotation layer* to describe its validity. As described in Section 2.2, the WCET of a program depends on its computer platform. The computer platform is typically divided into several layers, allowing the customization of the system at each layer. As shown in Figure 1 we propose to support the specification of at least the following three *annotation layers*:

Program Layer: If an annotation belongs solely to the program layer, the timing constraint is assumed to be platform-independent. Here it is important to note that in programming languages like C or C++ the functional behavior is not fully platform-independent, i.e., some timing constraints about the control flow may already belong to the *computer-platform layer*.

Computer-Platform Layer: The computer platform of a program includes everything necessary to execute the program. If a finer granularity is needed, the platform may be divided into different layers, like, for example, the build and run environment, the operating system, any middleware, and also the hardware (as shown in Figure 1.a).
For example, the cache geometry and the cache miss penalty may be specified at the hardware layer. As another example, knowing the attached flash memory device, one may specify the time needed for the completion of a write access.
Figure 1 also shows the difference between the orthogonal *layers* and the interface, a *platform* presents to a stack of layers. In Figure 1.a we see the different annotation layers, including the *computer-platform layers*, each of them clearly separated from the others. Please note the difference between a *computer-platform layer* (a name of an annotation layer) and a *platform* (as described in the MDA [14] of the OMG). In contrast to an annotation layer, a platform subsumes all the annotation layers below it. The platform can also be seen as an interface that comprises the information belonging to all

annotation layers below it. Thus, as shown in Figure 1.b, the system behavior influenced by each interface contains the behavior of all annotation layers below it.

Operation Layer: The *operation layer* describes the usage of the computer system, i.e., how the environment of the system is configured and how the environment behaves.
For example, timing constraints at the application layer may describe that the computer system is connected to three sensors, implying that a loop in the software to poll these sensors will iterate exactly three times.

The *program-*, *computer-platform-* and *operation-layers* are examples, only. Based on the specific system architecture, the user may refine the layering to further annotation layers. It can also happen that a timing constraint is associated to multiple annotation layers. However, whenever possible, it is advised to split such constraints into multiple constraints where each constraint belongs only to a single annotation layer. Note that the layer stack suggested by Figure 1.a is not mandatory; layers may be also placed horizontally. But the important point is that the different layers should be orthogonal, so that it is relatively easy in the system to exchange a layer and its specific timing annotations.

For timing constraints that refer to annotation layers other than the program layer, or timing constraints that represent *fictions*, more care has to be taken to ensure their intended use. For example, a loop bound may be tighter using information from the operation layer, as opposed to using only information from the program layer. Constraints refined with information from the operation layer are associated naturally also to the operation layer.

C1.2 Annotation Class

The annotation class is an attribute to describe the validity of timing constraints. As described in Section 2.3, besides the *timing invariants* we also introduce *timing fictions* as additional class of timing constraints. Each timing constraint should therefore contain a flag that indicates its class.

Invariants: *Invariants* are used to explicitly annotate information which is assumed to be valid with respect to the concrete semantics of the associated *annotation layer*.

Fictions: *Timing fictions* are used to provide fictive timing constraints to experiment with the sensitivity of a system's timing behavior.

The criterion of whether a timing constraint is an *invariant* (and not a *fiction*) is not only whether it holds for each possible input data on the program code. This is because, as shown in Figure 1.b, the system can be annotated at different layers (layers are described by the timing-constraint attributes **C1.1**).

For example, if a timing constraint describes properties of the *computer-platform layer*, we have to look at the concrete computer platform to decide whether this timing constraint is a *timing invariant* or a *timing fiction*.

C1.3 Annotation Group

The grouping mechanism allows for different WCET evaluations. For each annotation group a separate WCET calculation with its own set of timing constraints can be conducted.

There are several reasons why one might use different sets of timing constraints. For example, one might want to use and annotate different scenarios at the *operation layer*, or different tool chains at the *computer-platform layer*, etc. *Timing fictions* can be organized in groups as well to ensure their selective and correct use.

The grouping mechanism allows us to give each timing constraint membership to multiple groups. A group is a symbolic name together with a description field. There is no special semantics behind the groups: their intended meaning has to be described in their description fields. With the grouping

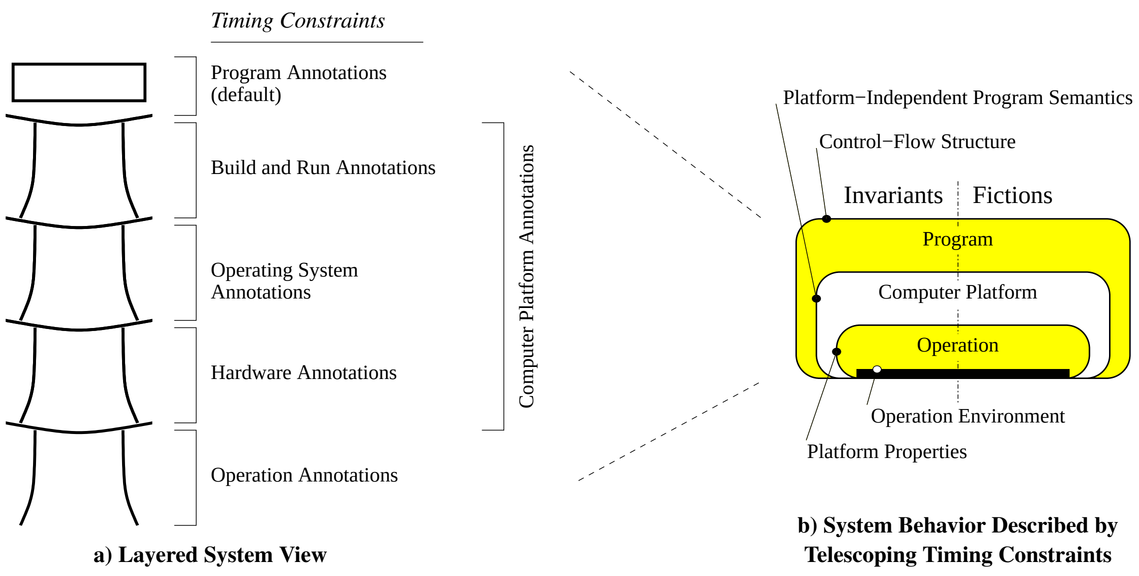

Timing Constraints
Program Annotations (default)
Build and Run Annotations
Operating System Annotations
Hardware Annotations
Operation Annotations
Computer Platform Annotations
a) Layered System View
Platform−Independent Program Semantics
Control−Flow Structure
Invariants
Fictions
Program
Computer Platform
Operation
Operation Environment
Platform Properties
b) System Behavior Described by Telescoping Timing Constraints

Figure 1. Layered Timing Constraints

mechanism one can specify which timing constraints will be used together for WCET analysis. Hierarchical definitions of groups are supported by specification of an optional list of nested groups.

Timing constraints that are *invariants* at the *program layer* are relatively easy to maintain. They can be checked directly against the source code and they only have to be changed if the program code changes. They remain valid if the computer platform changes.

C2 Program-Specific Annotations

We define program-specific annotations as timing constraints that directly describe the control and data flow of a program.

C2.1 Loop Bounds

Loop bounds comprise the minimal timing constraints at the *program layer* that are necessary to estimate the WCET of a program. For this reason, they were the first type of annotation that was introduced in the short history of WCET annotation languages [12].

Although loop bounds can always be expressed through linear flow constraints, there are practical reasons to allow loop bounds to be specified in a specialized and more compact notation. To maintain a tight execution count estimate after certain loop optimizations, it is desirable to specify lower loop bounds directly.

```
int i;
for (i = 0; i < n; ++i) {
  process(g[n]);
}
```

Here, the loop bound depends on the value of variable n. Static interprocedural program analysis over the whole program may find that the possible value of n at the beginning of the loop is 3...10, resulting in a lower loop bound of 3 and an upper loop bound of 10.

C2.2 Recursion Bounds

When a recursion is bounded, time and stack size requirements are also bounded using this recursion depth. If such conditions cannot be established by analysis, user annotations can supply the required data. In analogy to the earlier work on loop-bounds [1], Blieberger and Lieger established the conditions necessary for establishing upper bounds for stack space and time requirements of directly recursive functions. They also generalize the approach to indirectly recursive functions [2]. Recursion depth annotations are also used by Ferdinand et al. [4].

```
unsigned fac(unsigned n) {
  if (n == 0) return 1;
  else return n*fac(n-1);
}
```

The most precise recursion bound of procedure *fac* is the maximum value of input variable n. If a static program analysis finds *fac* always to be called with $n \leq 10$, then 10 is the most precise recursion bound.

C2.3 Linear Flow Constraints

Linear flow constraints are the basis for IPET-based WCET calculation methods. In the course of the calculation, all other program-specific constraints and control-flow constraints will eventually get translated into linear flow constraints. While flow constraints have a very high expressiveness, they are not necessarily as easy to write as, e.g., loop bounds, which is one of the reasons for allowing multiple ways of annotating the same flow constraint.

Linear flow constraints are used to express a relationship between certain reference points in the control flow graph (CFG) of a program. From the perspective of the source language this necessitates the introduction of auxiliary annotations like *markers* (to obtain a reference point) and *scopes* (to restrict the lexical validity of a constraint). The constraints themselves are usually called *restrictions*.

```
for (i = 0; i < n; ++i) {
  for (j = i; j ≥ 0; --j) {
    stmt1;
  }
}
```

We assume that the execution count of the entry of the outer loop is labeled as m_0 and the execution count of the inner loop's body is labeled as m_1. Then, the linear flow constraint "$m_1 \leq n \cdot (n-1)/2 \cdot m_0$" can be used to provide refined information about the execution count of the loop nest.

C2.4 Variable-Value Restrictions

Variable-value restrictions describe data-flow and are thus not a direct control-flow restriction. Variable-value restrictions can be transformed into an explicit control-flow restriction by a program analysis tool.

```
if (i < 72) {
  stmt1;
  ...
```

Directly before *stmt1* the value of i is confined by $i_{min} \leq i < 72$, where i_{min} is the smallest possible value of the data type of i.

C2.5 Summaries of External Functions

Often, software libraries are distributed as binaries and without any source code. In these cases, the library manufacturer could provide summaries of the library functions that contain the missing information that is necessary to analyze programs that use the library. A summary of a function may contain side effects (list of modified items) or value ranges of the returned values. A function summary may still be useful, even when the source code is available, e.g., for hard-to-analyze facts.

```
int signum(int x);
```

The subroutine *signum* is assumed to be pure and returns $-1, 0$ or $+1$. Thus we can annotate that the set of objects modified by this subroutine is empty, and the value returned by the subroutine is always from within $[-1, 1]$.

C3 Addressable Units

Addressable units of an annotation language are those that can be associated with timing constraints. The more language constructs and levels of abstraction can be addressed, the more fine-grained timing constraints can be specified. Examples of how to address different units of the program layer are given in [9]. In this section we list all language constructs that we consider relevant for being annotated with timing constraints.

C3.1 Control-Flow Addressable Units

Conceptually, WCET annotations typically express relationships between nodes, edges and paths of the CFG. If the paths between functions are included in the graph as well then we call this graph an interprocedural control flow graph (ICFG) [16]. Although the ICFG is implicitly defined by the program structure, it is not generally visible and will be generated ad hoc by the compiler.

The annotation language therefore faces the problem to address entities inside a graph that have no standardized explicit representation.

We thus propose the following addressable units of the ICFG based on the program source code:

C3.1a Basic Blocks

A basic block is a code sequence with single entry and single exit point. For timing analysis it is relevant that execution passes a basic block's entry point as often as its exit point. Thus, instead of annotating the basic block, any location within the basic block can hold the block annotation.

C3.1b Edges

Edges in the CFG, however, do not necessarily have a direct counterpart in the program because they are implicitly defined by the semantics of the respective language construct.

To circumvent this problem we introduce a set of reserved edge-names for each control flow construct of the source language. For example, considering some constructs of the C language, such names could include $TrueEdge_{if}$, $FalseEdge_{if}$ and $BackEdge_{while}$. Such names allow a user to associate timing constraints with specific edges of the respective CFG for a given language construct.

C3.1c Subgraphs

Subgraphs of the ICFG can be addressed and thus annotated. For example, an annotation can be associated with an entire function, or with a statement containing several function calls, or some nested loops.

To handle control flow inside expressions, such as function calls and short-circuit evaluation, it is necessary to normalize the program first. In this step short-circuit evaluation will be lowered into nested if-statements and function calls are extracted from expressions. For the addressing of subexpressions, a mapping between the normalized code and the original code must be maintained.

C3.2 Loop Contexts

For all kinds of loops it may be of interest to annotate specific iterations separately, or to exclude specific iterations, i.e. annotate all but these specific iterations. The most prominent example is that the first (few) iteration(s) may be very different from the following ones due to cache effects.

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < d; ++j)
    a[i][j] *= v[j];
```

Due to the "warming-up" of the cache, the first iteration could show a different behavior than the subsequent iterations.

C3.3 Call Contexts

As different call sites are bound to present different preconditions for a function e.g. input values, separate annotation of these different call contexts must be possible.

```
void g() { f(50); }

int f(int i) {
  while (--i ≥ 0) {
    ...
  }
```

The loop bound in function *f* depends on the value of input variable i. Thus, as a context-dependent flow constraint we can write that the upper loop bound is 50 when *f()* is directly called by *g()*.

C3.4 Values of Input Variables

If a function behaves significantly different depending on the possible values of an input parameter, it can be useful to provide different sets of annotations for each case. This kind of annotation was first introduced with SPARK Ada [15] and was called "modes".

```
int f(struct data *x) {
  if (x == NULL)
    return NULL;
  ...
}
```

The function may behave completely different depending on whether the input variable x is $NULL$ or not: e.g. whenever $x == NULL$, the function returns immediately.

C3.5 Explicit Enumeration of (In)feasible Paths

In path-based approaches [3, 7, 15, 18], explicit knowledge of the feasibility of paths can be incorporated into the analysis process.

```
void worker() {
  init();
  while (cond) process();
}
void process() {
  if (!initialized)
    init();
  ...
```

In this example, function *init()* is never called from function *process()*, if *process()* itself is called from function *worker()*. We can thus annotate that there is no path *worker→process→init*.

C3.6 The Goto Statement

The `goto` statement allows to introduce edges of non-structured control flow. If the target of a `goto` statement is statically known, it is not necessary to introduce any special annotations to specifically address a goto statement in the CFG; the containing basic block can be used equivalently. If the target address of a goto is not statically known, it makes sense to annotate possible jump targets as described in paragraph **C4.3**. The `break`, `continue` and `return` statements are specialized (better-behaved) instances of the `goto` statement in that their branch target is further restricted from function scope to the current control scope. This can be exploited by better analysis, but from the annotation standpoint there is not much difference to the `goto` except that there is less need for an annotation, when the analysis is easier.

C4 Control-Flow Constraints

The CFG is a valuable abstraction level that can be refined in various ways to improve the precision of the analysis. This is to aid the automatic CFG generation within the tools by additional information that is not available within the program itself.

C4.1 Unreachable Code

This is a program-specific annotation, which has been used by Heckmann and Ferdinand [8]. Unreachable code could as well be specified by linear flow constraints. Having a specific mechanism however makes the intention of the user explicit.

C4.2 Predicate Evaluation

Closely related to the above case, annotations of predicate evaluations were also introduced by Heckmann and Ferdinand [8]. These kind of annotations describe for conditions/decisions whether they will always evaluate to True or False.

C4.3 Control-Flow Reconstruction

Introduced by Ferdinand et al. [5], and further elaborated by Kirner and Puschner [13], the CFG Reconstruction Annotations are used as guidelines for the analysis tool to construct the control flow graph (CFG) of a program. Without these annotations it may not be possible to construct the CFG from the binary or object code of a program.

On one hand, annotations are used for the construction of syntactical hierarchies within the CFG, i.e., to identify certain control-flow structures like loops or function calls. For example, a compiler might emit ordinary branch instructions instead of specific instructions for function calls or returns. In such cases it might be required to annotate a branch instruction whether it is a call or return instruction.

The high-level programming language features that can lead to code that is difficult to analyze locally are: function-pointer calls, virtual-method calls, and returns as well as indirect conditional control-flow transfer like computed goto or switch statements or transformation results obtained from combining conditional control flow with ordinary or indirect calls or returns.

```
void process((void)(int*) func,
        int *data) {
  (*func)(data);
}
```

In this code, it might be known that the target of function pointer *func* points either to `(void)reset(int*)` or to `(void)iterate(int*)`.

A work-around that sometimes helps avoiding code annotations is to match code patterns generated by a specific version of a compiler. However, such a “hack” cannot cover all situations and may also have the risk of incorrect classifications, for example, if a different version of the compiler is used.

On the other hand, annotations may be needed for the construction of the CFG itself. This may be the case for branch instructions where the address of the branch target is calculated dynamically. Of course, static program analysis may identify a precise set of potential branch targets for those cases where the branch target is calculated locally. In contrast, if the static program analysis completely fails to bind the branch target, it has to be assumed that the branch potentially branches to each instruction in the code, which obviously is too pessimistic in order to compute a useful WCET bound. In such a case, code annotations are required that describe the possible set of branch targets.

The following list summarizes examples of code annotations derived from aiT [5, 8]:

- `instruction <addr> calls <target-list>;`
- `instruction <addr> branches to <target-list>;`
- `instruction <addr> is a return;`
- `snippet <addr>     is never executed;`
- `instruction <addr> is entered with <state>;`

Note that these annotations need not be linked to a specific instruction type, since an optimizing compiler may transform

```
call F          push L ; prepare a return to a different address
jump L   into:  jump F ; jump to function, return to target
```

This is also known as triangle call or triangle jumps. Now the jump instruction represents the logical call followed by the jump and must bear both annotations.

C5 Hardware-Specific Annotations

For a realistic modeling of the execution behavior of a program, an annotation language also needs mechanisms to describe the behavior of the underlying hardware. Many of these annotations are supported by industrial timing analyzers like aiT [8].

Since some hardware-specific annotations are associated to the *hardware layer* only, they are independent from the program layer and can thus be easily reused for multiple programs running on the same embedded platform. It can thus make sense *not* to annotate this information to program code, but rather gather it in a common location so that it can be combined with the annotations of more than one program.

Examples of such basic hardware data to be kept separate from the program annotations are:

Instruction timing: The general timing information of instructions has to be maintained separate from the program.

Clock rate: The analysis must be able to convert clock ticks to absolute times when computing the WCET, and vice versa for absolute-time specification annotations.

Access times for ROM, internal and external RAM: It would be tedious and cumbersome to specify these times at each of the various read and write operations.

Memory map: As the memory map binds memory access times to a multitude of memory access operations, the information that is available to the linker can, when supplied to the timing analysis, largely reduce the annotation effort for the program.

Hardware implementation details that hold on the program as a whole, and cannot be tied to a single specific program location, also need to be specified separately. Caches or jump prediction details are examples.

It is not always obvious where to draw the borderline between hardware-specific annotations and information that is better managed by the analysis tool. The following items are examples of timing constraints that are reasonably expressed as timing annotations.

C5.1 Memory and Memory Accesses

The temporal behavior of memory accesses depends on the characteristics of the memory. Embedded systems typically use different types of memory depending on the access frequency and access pattern. It is thus necessary to specify the following characteristics:

- address range of read operations
- address range of write operations
- writeable memory area (e.g., RAM, Flash-ROM) and read-only memory area (ROM)
- data and code regions
- access time of specific memory regions (in cycles or ms)

C5.2 Absolute Time Bounds

Providing a means for absolute time bounds allows to specify the maximum and minimum execution time of a fraction of code. Such a feature can be found in WCETC [11], for example.

```
char poll() {
  volatile char io_port;
  while (io_port ≠ 0)
    /* wait */ ;
}
```

It could be an invariant of the hardware platform that the execution time of the subroutine *poll()* (busy waiting) is always between 30 and $100\mu s$.

4. Conclusion

The lack of common interfaces and open analysis frameworks is an impediment for the research in WCET analysis. Activities have been started within the ARTIST2 Network of Excellence to define such a common WCET analysis platform. As part of this, *The Annotation Language Challenge* for WCET analysis has been proposed [12]. This paper is aimed to be a first step towards a *common WCET annotation language*. It describes essential ingredients such an annotation language should include. The timing constraints are described conceptually to allow instantiation for different representation levels and tools.

We analyzed existing timing-annotation constructs and described them in a conceptual way. We identified the potential need for further mechanisms and developed some new ingredients for annotation languages. Among the new contributions are the categorization techniques of timing constraints by the separation between *timing invariants* and *timing fictions*, the introduction of *annotation layers*, *annotation groups*. Further, we gave a discussion of *addressable units* to be used for annotating the program.

We consider the proposed list of essential ingredients for a WCET annotation language as complete for procedural languages. Therefore we want to encourage professionals and researchers to provide their feedback as a basis for the refinements of this list.

Acknowledgments
We would like to thank Niklas Holsti from Tidorum Ltd for his valuable comments on this paper.

References

[1] Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[2] Johann Blieberger. Real-time properties of indirect recursive procedures. *Inf. Comput.*, 171(2):156–182, 2001.

[3] Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[4] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. of the 1st Int'l Workshop on Embedded Software (EMSOFT 2001)*, Tahoe City, CA, USA, Oct. 2001.

[5] Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003.

[6] Jan Gustafsson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.

[7] Christopher A. Healy and David B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions of Software Engineering*, pages 763–781, Aug. 2002.

[8] Reinhold Heckmann and Christian Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.

[9] Niklas Holsti. *Bound-T Assertion Language*. Space Systems Finland Ltd, Espoo, Finland, 6.2 edition, Feb. 2008. online available at: `http://www.tidorum.fi/bound-t/assertion-lang.pdf`.

[10] IST-004527. The ARTIST2 Network of Excellence on Embedded Systems Design. `http://www.artist-embedded.org/`, September 1st 2004 - August 31st 2008. ARTIST2 is funded by the European Commission within FP6.

[11] Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[12] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET analysis: The annotation language challenge. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.

[13] Raimund Kirner and Peter Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.

[14] Object Management Group. *MDA Guide*, version 1.0.1 edition, June 2003. document number: omg/2003-06-01.

[15] Chang Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[16] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981. ISBN:0137296819.

[17] Alexander Vrchoticky. Modula/R - Language Definition. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Mar. 1992.

[18] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.

[19] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckman, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), Apr. 2008.

TOWARDS AN AUTOMATIC PARAMETRIC WCET ANALYSIS[1]

Stefan Bygde and Björn Lisper
School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{stefan.bygde, bjorn.lisper}@mdh.se

Abstract
Static WCET analysis obtains a safe estimation of the WCET of a program. The timing behaviour of a program depends in many cases on input, and an analysis could take advantage of this information to produce a formula in input variables as estimation of the WCET, rather than a constant. A method to do this was suggested in [12]. We have implemented a working prototype of the method to evaluate its feasibility in practice. We show how to reduce complexity of the method and how to simplify parts of it to make it practical for implementation. The prototype implementation indicates that the method presented in [12] successfully can be implemented for a simple imperative language, mostly by using existing libraries.

1. Introduction

Most state-of-the-art static WCET analyses derive a constant time estimation of the WCET of a program. In the general case, this estimation has to be pessimistic in order to be safe. However, many tasks are re-configurable and/or reused in different applications and the worst case derived by the analysis applies only to one configuration/input. The real WCET for a fixed configuration or input data might be significantly tighter than the global worst case. One analysis could be made for each configuration or input data, and new analyses can be made as needed. This, however, is tedious, inflexible and requires that the analysis tools are available whenever the data or configuration of the program is changed. In [12] a static analysis which provides a safe parametric estimation of the WCET as a formula of input variables rather than as a constant value is presented. We have done a prototype implementation of this analysis and we will present some of the technical issues that arise in a practical implementation. The reason for the implementation is to test the feasibility of the analysis framework in practice. The contributions of this paper are

- Showing that the analysis can be implemented for simple programs (see Section 2)
- Reducing complexity of the analysis by reducing the number of variables
- Simplifying the method in [15] for counting solutions of presburger formulae to count points inside convex polyhedra
- Identifying the need for simplification of the resulting parametric WCET formula

[1]This work is supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS

Section 2 briefly summarises the method given in [12] and in Section 3 we show how we have implemented it. Section 4, shows how to simplify symbolic counting of the number of solutions based on [15]. Then, in section 5, we show how to reduce the complexity of the problem by eliminating variables from the calculation phase. In Section 6, we discuss the problem with a big and complex solution and suggestions on how to solve it. Finally we present related work in Section 7 and conclusions and future work in Section 8.

2. Method

We first introduce some terminology and make a few assumptions. By *program* we shall mean a small program, a task or a function which has a well defined set of input parameters. Furthermore, we assume that the analysed program is terminating and deterministic, and that variables that affect the program flow are integers or Booleans. We will refer to *program points* as the edges of the control flow graph (CFG). The analysed program will be denoted P, and its set of program points $\mathcal{Q}_P$. We shall assume that any program uses a set of variables $\mathcal{V}_P$ and that each program has a set of *input parameters* $\mathcal{I}_P \subseteq \mathcal{V}_P$ which has two properties: 1) They affect the program flow and 2) They are not changed during execution of P. As long as these properties are fulfilled the set of input variables can be chosen arbitrarily. Now we will for completeness summarise parts of the method described in [12].

The method assumes that there exists an analysis producing structural constraints[2] of the program and a low-level analysis that produces a WCET estimation for all atomic parts (program points) of the program. The idea here is to have the execution count of each program point bounded from above by a parameter and to perform an parametric IPET calculation. Other constraints, such as constraints on paths can also be bounded by parameters. The result of the parametric IPET calculation is a function expressed as a formula

$$\mathrm{PC}_P : \mathbb{N}^{|\mathcal{Q}_P|} \to \mathbb{N}$$

where the domain is a vector where each component represents the maximum bound on the execution count for a program point. We call the parameters in this vector *execution bound parameters*. Furthermore, we call the function PC_P *Parametric Calculation*. The maximum execution count for individual program points is often dependent on input parameters. The arguments of PC_P will therefore be computed in terms of $\mathcal{I}_P$. Formally, the arguments will be computed by a function

$$\mathrm{MEC}_P : \mathbb{Z}^{|\mathcal{I}_P|} \to \mathbb{N}^{|\mathcal{Q}_P|}$$

where the domain vector is an instantiation of the input parameters. We call this function the *Maximum Execution Count* function. When the parametric calculation and the maximum execution count functions have been computed, we can obtain the WCET estimation as a parametric formula in the input variables as the function

$$\mathrm{PWCET}_P : \mathbb{Z}^{|\mathcal{I}_P|} \to \mathbb{N} = \mathrm{PC}_P \circ \mathrm{MEC}_P\ .$$

The analysis boils down to finding the functions PC_P and MEC_P for a given program P. The function PC_P can be obtained by performing parametric IPET calculation using parametric ILP [9]. The MEC_P function can be obtained by observing the fact that in a deterministic, terminating program, each run-time state has to be unique[3]. Thus, the number of times a program point can be visited

[2]Constraints imposed by the graph structure of the CFG only

[3]If exactly the same state is encountered twice, the determinism of the program enforces the program to repeat the same sequence of states infinitely.

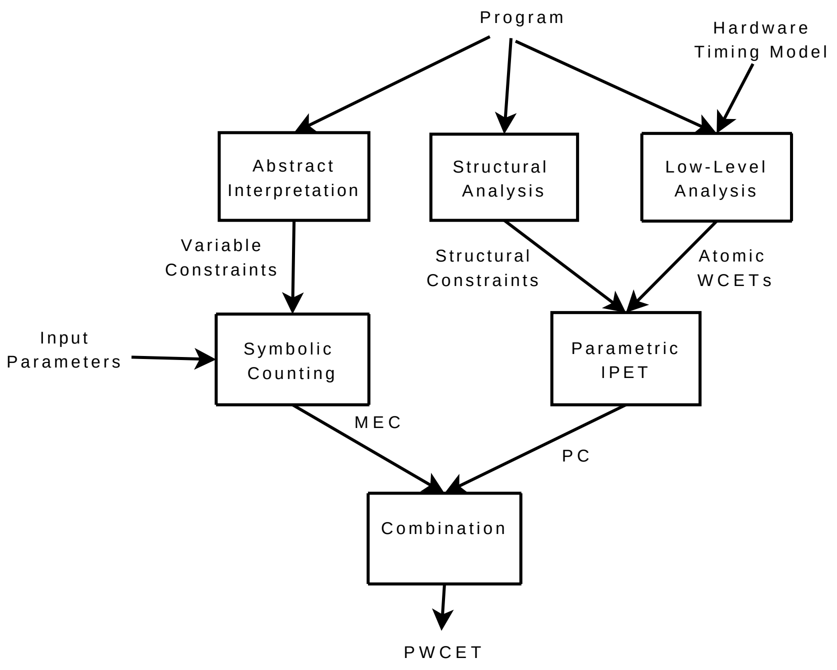


Figure 1. Work flow

is upper bounded by the number of distinct run-time states. Abstract Interpretation [6] can derive a super set of the set of possible run-time states at each program point. A relational domain is an abstract domain that preserves some of the relations among variables in the analysis. If the abstract domain used in the abstract interpretation is relational, then some of the variables can be used to parameterise the set of possible run-time states w.r.t. other variables. The maximum execution count function can then be obtained by counting the number of states contained in the solutions of the abstract interpretation with respect to the parameters. The work flow is shown in Figure 1.

3. Implementation

We have made a prototype implementation of the analysis presented in the last section. We represent programs as CFGs where nodes are start, stop, assignment or conditionals. Furthermore, all variables are assumed to be integers or Booleans. We do not consider function calls or pointers and the expressions in assignments and conditionals have to be linear. The reason for this restriction is that most relational abstract domains can only capture linear relations among variables. Since the computational tasks required to realise this are quite modular, we have implemented different parts independently of each other and mostly by reusing existing code and libraries. We present for each box in Figure 1 the corresponding implementation, except for structural analysis and low-level analysis which are assumed to exist. In our prototype we simply assume that all atomic parts of the program have the same constant WCET of ten clock cycles, but this could easily be replaced by the real results of a low-level analysis. Since we represent our program as a CFG, we can easily derive the structural constraints for each node by requiring that its in-flow should be equal to the out-flow.

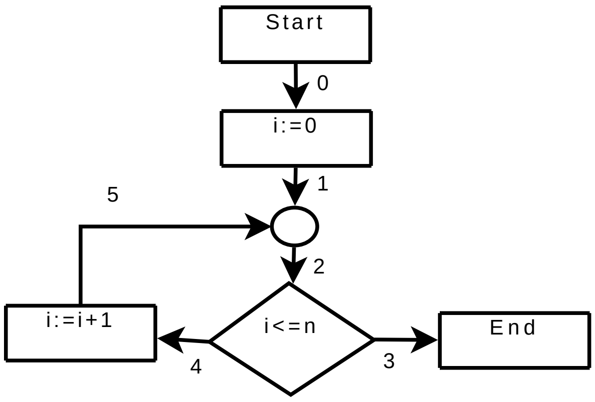


Figure 2. Example program L where all program points have been labeled

3.1. Abstract Interpretation

The main design decision of the abstract interpretation is the choice of abstract domain. The requirements of the abstract domain is that, in order to be parametric, it should be relational and that the domain can derive bounds on the values (to be able to count states). We have implemented the abstract interpretation with the polyhedral domain [7] using the new polka library [13]. Such an abstract interpretation can derive linear constraints among variables, enclosing the possible values of the program variables at a program point inside a convex polyhedron in n dimensions, where n is the number of variables. A simple framework for abstract interpretation for our CFGs has been implemented in C++. The implementation allows support for other abstract domains to be added with little effort. The polyhedral domain has exponential complexity in the number of dimensions (in this case, variables) but has a quite good precision. As an example, consider the CFG of a program we call L in Figure 2. Performing polyhedral abstract interpretation will yield a vector $\vec{a}$ of abstract values for each program point as follows:

$$\begin{array}{ll} a_0 = \top & a_3 = \{i \geq 0, i \geq n+1\} \\ a_1 = \{i = 0\} & a_4 = \{0 \leq i \leq n\} \\ a_2 = \{i \geq 0\} & a_5 = \{1 \leq i \leq n+1\} \end{array} \tag{1}$$

where $\top$ means "no information".

3.2. Counting States

To obtain the MEC_P function from the result of the abstract interpretation, we need to count the number of integer solutions for each system, parameterized in the set of chosen input parameters $\mathcal{I}_P$. We have implemented a simplification of the method in [15]. The method counts the number of solutions to a given presburger formula, of which a system of linear inequalities is a subset. We describe our modelling and simplification of the method in Section 4. The result from the symbolic counting represent the function MEC_P. For completeness we now sketch parts of the method in [15], it computes the result of generalised sums $(\Sigma V : P : x)$ where V is a set of variables to sum over, P is a presburger formula (the guard) and x is any formula. The result of such a sum is the sum for all variables $v \in V$ which satisfy P of x. As a simple example, $(\Sigma\{v\} : l \leq v \leq u : v)$ is equal to $\sum_{v=l}^{u} v$. The general method to compute $(\Sigma V : P : x)$ is to choose a variable $v \in V$ and compute the general sum

$$(\Sigma V \setminus \{v\} : P' : (\Sigma\{v\} : l \leq v \leq u : x))$$

where P' is P where all information about v is removed. Since $(\Sigma\{v\} : l \leq v \leq u : x)$ is equivalent to $\sum_{v=l}^{u} x$, known formulae of summations over the form of x can be used to simplify it. If $V \setminus \{v\}$ is non-empty another variable is chosen and the procedure is repeated until $V = \varnothing$, and the result is a sum of generalised sums $(\Sigma : G : x')$ which should be read as "x' if G holds, else 0". This result is symbolic in the variables occurring free in x or P but not in V. The situation is however not always this easy; variables can have several lower/upper bounds or be unbounded and bounds can be negative and/or rational. All these cases are handled in [15].

Given a vector $\vec{a} = (a_0\ a_1\ ...\ a_{m-1})$, with $m = |\mathcal{Q}_P|$ components, where each component is a convex polyhedron obtained from the abstract interpretation, we define the image through MEC_P of the k-th component as

$$\mathrm{MEC}_P(\vec{i})_k = (\Sigma(\mathcal{V}_P \setminus \mathcal{I}_P) : a_k : 1)$$

for $k = 0...m-1$. The right hand side is a formula of the elements in $\mathcal{I}_P$ which are instanced with the elements of $\vec{i}$. Consider the CFG L in Figure 2 and the result $\vec{a}$ from the abstract interpretation in (1). For L we have $\mathcal{V}_L = \{i, n\}$ and $\mathcal{I}_L = \{n\}$ so by symbolically counting the number of integer points inside the polyhedra of will give the MEC_L function as follows (we will here write Σi for $\Sigma\{i\}$).

$$\begin{aligned}
\mathrm{MEC}_L(n)_0 &= (\Sigma i : \varnothing : 1) = \infty \text{ (unbounded sum)}\\
\mathrm{MEC}_L(n)_1 &= (\Sigma i : i = 0 : 1) = 1\\
\mathrm{MEC}_L(n)_2 &= (\Sigma i : i \geq 0 : 1) = \infty\\
\mathrm{MEC}_L(n)_3 &= (\Sigma i : i \geq 0, i \geq n+1 : 1) = \infty\\
\mathrm{MEC}_L(n)_4 &= (\Sigma i : 0 \leq i \leq n : 1) = (\Sigma : n \geq 0 : n+1)\\
\mathrm{MEC}_L(n)_5 &= (\Sigma i : 1 \leq i \leq n+1 : 1) = (\Sigma : n \geq 0 : n+1)
\end{aligned} \tag{2}$$

Now MEC_L maps the value of n to the number of possible states (number of points inside polyhedra) at each program point of the program.

3.3. Parametric IPET

Parametric Integer Programming [9] is essentially a parameterized version of the (dual) simplex algorithm and may be used to solve parametric IPET problems. A tool called PipLib exists [14], it can be used to solve a parametric IPET problem, where the answer is given in terms of the parameters. With structural constraints $A\vec{x} \leq \vec{b}$ and atomic WCETs from the low-level analysis $\vec{c}$, we can use PipLib to maximise

$$\vec{c}^{\mathrm{T}}\vec{x} \text{ subject to } A\vec{x} \leq \vec{b} \text{ and } \vec{x} \leq \vec{p}$$

where $\vec{p}$ is a vector of parameters where each component p_i corresponds to a symbolic upper bound to the execution count of x_i. PipLib gives as solution a so-called *quast*, which is a formula expressing the unknown variables in terms of the parameters. The quast is a tree of nested if-statements where the leaves correspond to solutions for the unknown variables. Both the conditionals and the solutions are linear expressions of the parameters, see [14] for further information. The quast can be used to obtain PC_P.

If we assume that each program point of L in Figure 2 has a constant WCET of ten clock cycles then we want to maximise $y = 10 \sum x_i$ subject to the structural constraints of L and $x_i \leq p_i$. Using PipLib to solve this IPET problem will give us PC_L:

$$\mathrm{PC}_L = \lambda(p_0, p_1, p_2, p_3, p_4, p_5).\mathrm{if} p_2 \leq p_4 + 1 \text{ then}(\mathrm{if} p_2 \leq p_5 + 1 \text{ then } 30p_2 + 10 \text{ else}... \tag{3}$$

The full quast contains eight leaves and is too big to show here. The reason for the many leaves is that the possible relations among the parameters are many.

3.4. Combination

Finally, we need to express the WCET estimation as a parametric formula in terms of the input parameters in $\mathcal{I}_P$, formally computing the function $\mathrm{PWCET}_P = \mathrm{PC}_P \circ \mathrm{MEC}_P$. This is done by parsing the solution quast given from PipLib (corresponding to PC_P) and substituting the parameters $\vec{p}$ with $\mathrm{MEC}_P(\vec{i})$, resulting in a formula parameterized in $\mathcal{I}_P$.

The final result of the analysis for L will after combining (3) and (2) is,

$$\mathrm{PWCET}_L = \lambda n.\mathrm{if} c_2 \leq c_4 + 1 \text{ then(if } c_2 \leq c_5 + 1 \text{ then } 30c_2 + 10 \text{ else...}$$

where $c_k = \mathrm{MEC}_L(n)_k$. Since $c_5 = c_4$ and $c_1 = 1$ we can see that simplification is needed, but not yet implemented.

4. Symbolic Counting

The method in [15] gives a symbolic formula of the number of solutions to a given presburger formula. Convex polyhedra (or equivalently, systems of linear inequalities) are a strict subset of presburger formulae and we will show to restrict and simplify the method for this special case. We will assume two restrictions of the generalised sums; first, rather than having the guard as a presburger formula, we have it as a system of linear inequalities in the variables of $\mathcal{V}_P$ (since this is exactly what the polyhedral abstract interpretation will give). The other restriction is that we model the formulae to sum over as polynomials, simplifying both representation and computation. Polynomials can easily be modelled as a sum of terms, where a term is a vector representing an integer coefficient and variable powers. As an example we can model the polynomial $3a^2b^3 + 5a^4$ (assuming $\mathcal{V}_P = \{a, b\}$) as the sum of the terms $(3\ 2\ 3)$ and $(5\ 4\ 0)$. This also makes arithmetical operations on these vectors straightforward to implement. Furthermore, since the guards are polyhedra, the lower and upper bound of any variable will be sets of linear expressions. Summing a linear expression over a polynomial is again a polynomial, so this model is closed under summations. However, these restrictions sometimes require the result to be slightly over-approximated. The constraint $3a - b \leq 0$ gives an upper bound for a as $a \leq \lfloor \frac{b}{3} \rfloor$, since a and b are integers. As seen, the upper bound is not a polynomial and therefore problematic in our model. Since $\frac{b}{3}$ is a safe upper bound for $\lfloor \frac{b}{3} \rfloor$ and on polynomial form we can use it as approximation. We handle lower bounds accordingly.

5. Reducing the Number of Variables

As seen above, we need an unknown variable and an execution count parameter for each program point in the program in the parametric IPET calculation. Parametric ILP has exponential complexity in the number of variables in the worst-case, making scalability problematic. The structural constraints of the program in general produce an under determined system. In an under determined system with n variables, the solution space is the span of a set of $n-r$ vectors (where r is the rank of the constraint matrix). The variables can be expressed as linear combinations of these vectors and we can only solve the problem in terms of these. Let $A\vec{x} = \vec{b}$ be a system of structural and possible other linear equations obtained from flow analyses and $y = \vec{c}^{\mathrm{T}}\vec{x}$ be the cost function. The constraints together with the cost function is

$$\begin{pmatrix}1 & -\vec{c}\\ 0 & A\end{pmatrix}\begin{pmatrix}y\\ \vec{x}\end{pmatrix} = \begin{pmatrix}0\\ \vec{b}\end{pmatrix}$$

If we perform Gauss-Jordan elimination on the above (including the right-hand side by augmenting the constraint matrix by $(0\ \vec{b})^{\mathrm{T}}$) and re-arrange the columns of A and the components of $\vec{x}$ such that all pivot columns are to the left, and $\vec{x}$ is re-arranged accordingly, we get

$$\begin{pmatrix}1 & 0 & -\vec{c}'\\ 0 & I_r & A'\end{pmatrix}\begin{pmatrix}y\\ \vec{x}_{\mathrm{BV}}\\ \vec{x}_{\mathrm{FV}}\end{pmatrix} = \begin{pmatrix}z\\ \vec{b}'\end{pmatrix}$$

where I_r is the $r \times r$ identity matrix and r is the rank of A. Furthermore, z is the last column of the solution of the augmented matrix after elimination. The vector $\vec{x}$ has now been partitioned into a vector of r basic variables $\vec{x}_{\mathrm{BV}}$ which corresponds to the columns of I_r and the vector $\vec{x}_{\mathrm{FV}}$ of $n - r$ free variables (where n is the number of columns of A) which corresponds to the columns of A'. Note that this transformation also removes any redundant constraints from the system. From this we can derive two important equations. One is the objective function expressed in terms of the free variables

$$y = z + \vec{c}'\vec{x}_{\mathrm{FV}}$$

and a way to express the basic variables in terms of the free ones

$$\vec{x}_{\mathrm{BV}} = -A'\vec{x}_{\mathrm{FV}} - \vec{b}'$$

As we model the parametric upper bounds as the constraints $\vec{x} \leq \vec{p}$, we can now simply model our IPET problem as

$$\begin{pmatrix}z + \vec{c}'\vec{x}_{\mathrm{FV}}\\ -A'\vec{x}_{\mathrm{FV}} - \vec{b}'\\ \vec{x}_{\mathrm{FV}}\end{pmatrix} \leq \begin{pmatrix}y\\ \vec{p}_{\mathrm{BV}}\\ \vec{p}_{\mathrm{FV}}\end{pmatrix}$$

where we have partitioned and re-arranged $\vec{p}$ exactly as for $\vec{x}$. Now it suffices to solve the IPET with these constraints, thus reducing the number of unknowns by the rank of A. Note that we cannot reduce the number of parameters in the same way, since they directly correspond to the output of MEC_P. This method of eliminating variables is not restricted to the parametric case, but can be used to reduce the dimensionality of any IPET problem. As an example, using this on (3) reduces the system from eight to two unknown variables (the variable of the cost function included).

6. The Result of Parametric Integer Programming

The possible number of relations between parameters in the IPET calculation are large, and yields large and complex quasts as result, even for small program examples (as seen in Section 3.3). Calculating WCET_P as a composition of the quasts with the result of the symbolic counting will substitute guarded generalised sums for all parameters in the quasts and the final result will be very complex and large. This suggests that a simplification of PC_P is needed as well as of the final formula PWCET_P. As presented in Section 5, we already simplified the problem by reducing the number of unknowns, but there are more possibilities for simplifications. A way to reduce the number of possible relations among parameters would be to restrict the parameters in the IPET calculation with constraints. Restrictions of parameters could potentially be found directly from MEC_P (e.g. some counts may impose absolute upper and lower bounds on parameters) and in some cases from the CFG structure (e.g. start and stop nodes restrict the parameters). Some parameters could also potentially be eliminated. The final PWCET_P obtained after substitution can be simplified by finding equal sub trees and eliminating tautologies and contradictions in the conditionals.

7. Related Work

The analysis in [8] is as [12] based on calculating the number of run-time states to find the WCET but without parameters. An approach to parametric WCET analysis was presented in two joint master theses [1, 11] and also in [2]. The analysis of these theses uses PipLib as tool for a parametric IPET calculation but uses parameters only for loop bounds. The loop bound parameters is then substituted for parameterized intervals. The parameterized intervals are obtained by identifying loop counter variables and loop invariants based on their relative values to input parameters of the program. Other approaches to parametric WCET analysis can be found in [3, 5, 16].

Abstract interpretation [6] is a general theory for soundly abstracting program semantics and is commonly used in static analysis. The polyhedral domain [7] has been used in different contexts than this, such as in verification of linear hybrid systems in [10]. Different methods for symbolic counting are [15, 4]. Parametric integer linear programming was first presented in [9], see also [14].

8. Conclusions and Future Work

We have done a prototype implementation of the framework for parametric WCET analysis presented in [12]. The framework was implemented for simple CFGs. The prototype is in the current state not powerful enough to fully evaluate the method, but shows that the method successfully can be implemented, mostly by using existing libraries. We have presented a simplified method for symbolically counting integer points inside convex polyhedra based on the method in [15]. We have also presented a general method to reduce the number of variables used in a IPET calculation where linear equations are used as constraints. We conclude that an important challenge of making a successful implementation for real programs is to deal with complexity; both computational and representational. The final WCET formula needs to be simplified in several steps. As future work, the prototype will be enhanced to be able to deal with more realistic program (by adding arrays and pointers) in order to evaluate and make comparative studies on concrete benchmarks and to compare the precision/complexity trade-off by using different abstract domains and calculation methods.

References

[1] S. Altmeyer. Parametric wcet analysis, parametric framework and parametric path analysis. Master's thesis, Saarland University, Department of Computer Science, Oct 2006.

[2] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. Parametric timing analysis for complex architectures. In *RTCSA '08: Proc. 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.*, 2008.

[3] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proc. 25th Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[4] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.

[5] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analysis. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 152–154, New York, NY, USA, 2007. ACM.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[7] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[8] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. 7^{th} International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, July 2007.

[9] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[10] Nicholas Halbwachs, Yann-Eric Proy, and Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In Baudouin Le Charlier, editor, *Proc. International Symposium on Static Analysis*, Vol. 864 of *Lecture Notes in Comput. Sci.*, pages 223–237, Namur, September 1994. Springer-Verlag.

[11] C. Humbert. Parametric wcet analysis, parameter analysis and parametric loop analysis. Master's thesis, Saarland University, Department of Computer Science, Oct 2006.

[12] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, pages 77–80, Porto, July 2003.

[13] New polka webpage, 2008. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/newpolka/index.html`.

[14] Piplib website, 2008. `http://www.piplib.org/`.

[15] William Pugh. Counting solutions to Presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[16] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In Jay Fenwick and Cindy Norris, editors, *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 88–93, Snowbird, Utah, June 2001.

TOWARDS PREDICATED WCET ANALYSIS

Amine Marref, Guillem Bernat[1]

Abstract
In this paper, we propose the use of constraint logic programming as a way of modeling context-sensitive execution-times of program segments. The context-sensitive constraints are collected automatically through static analysis or measurements. We achieve considerable tightness in comparison to traditional calculation methods that exceeded 20% in some cases during evaluation. The use of constraint-logic programming in our calculations proves to be the right choice when compared to the exponential behaviour recorded by the use of integer linear-programming.

1. Introduction

WCET analysis have been explored for about two decades and can be divided into three categories: end-to-end testing, static analysis (SA), and measurement-based analysis (MBA) [6]. SA and MBA finds the WCET of a program as follows: (a) decomposing the program into segments, (b) finding the execution times of these segments, and (c) combining these execution times using a calculation technique: tree-based [5], path-based [8], or implicit path-enumeration (IPET) [9, 14]. Path-based methods suffer from exponential complexity and tree-based methods cannot model all types of program-flow, leaving IPET as the preferred choice for calculation because of the ease of expressing flow dependencies and the availability of efficient *integer linear-programming* (ILP) [1] solvers.

Current calculation techniques struggle to cope with variations in execution times of program segments caused by modern-hardware speed-up features because of the complexity resulting from modeling all these timing variations. This motivates the use of a more powerful calculation technique which copes with execution time variations and yields tighter, more context-sensitive WCET estimations.

We proceed by identifying the necessary conditions leading to the observation of different execution times of program segments. These conditions are expressed as implications which by definition are disjunctions (if a and b are predicates than $(a \Rightarrow b) \equiv (\neg a \vee b)$). There can be many segments which have multiple execution times, and each execution time of the segment is caused by one or more segments that previously executed. This makes the total number of constraints to handle considerably large.

In the current work, we use *constraint-logic programming* (CLP) [2] in order to express the constraints governing the execution flow and times of the segments in the program. All constraints including implications/disjunctions can be encompassed in the same model using CLP and with no model expansion. These two features make CLP solve an IPET model within seconds, which is otherwise solved using ILP in hours because of model duplication (ILP handles disjunction through model duplication). CLP also enables the integration of execution-time analysis of many hardware components (Section 6), an issue that has never been properly resolved (Section 2).

[1]Department of Computer Science, University of York, Heslington, YO105DD, UK

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 establishes the required terminology. Section 4 explains the deriving of constraints required for CLP calculation by either static analysis or through measurements. Section 5 explains how the constraints are expressed in CLP and highlights the problems encountered when trying to express the constraints in our work using ILP. Section 6 shows an example where constraints derived from many hardware components analysis are integrated together. Section 7 shows the tightness in WCET estimations we obtained during the evaluation of our context-sensitive calculation. Finally, Section 8 summarizes the important results and sets aims for future work.

2. Related Work

WCET analysis using context-sensitive IPET is not novel, the idea has been around for a long time. What is common in the literature is that execution-context could not be fully exploited because of the exponential complexity that usually accompanies the process. In addition, the focus has always been to achieve context sensitivity in the execution counts of segments rather than the execution times due to the fact that ILP is used to compute the solution which is by definition linear (i.e. either execution counts or execution times can vary but not both).

There has been much work on WCET calculation using IPET starting in [9, 14] where the execution times are constant. In later work e.g. [10, 4], ILP was used to represent more execution history by modeling caches, pipelines, branch predictors, and speculative execution. The objective function is generally augmented by execution-time gains/penalties resulting from the use of the hardware component being analysed. These objective functions have not been integrated together because of their complexity. Each segment has two execution times at most.

In a very recent work [16], timing variability of basic blocks with respect to pipelines has been analysed where there can be multiple execution times of a basic block depending on the subpath previously traversed, then ILP is used to calculate the overall WCET.

By and large, the work in [7] is the most related to ours. In [7] WCET calculation is performed using the notion of scopes to provide context-sensitivity mainly through constraining execution counts. At the low-level, the execution-times of segments are expressed in a scenario-based fashion where multiple execution times are allowed in theory. The integration in the ILP model requires bounds on different execution times of a basic block to be known a priori which as we shall see (Section 5.2) still generates pessimism and are very hard to derive. The work in [7] assumes that the constraints are provided by a user, so although the calculation is context-sensitive, its practical value is limited to when such constraints are available. The low-level constraints such as “block B_1 has got execution time 100 at most six times” are very unlikely to be provided by the user and must be derived automatically which is hard. This leads to the fact that the evaluation incorporated only the pipeline effects of a block over its immediate predecessor i.e. a block has at most two execution times.

Our work is different firstly in the sense that it allows all conditional execution-times to be expressed easily so there is no need for akwardly written ILP objective functions. The constraints governing the observation of execution times of a block are determined automatically without the need to put bounds on the number of such observations which are then computed exactly using CLP. Secondly, all constraints with respect to all hardware components are integratable together by simple conjunctions. Thirdly, solving these constraints using CLP takes seconds while it can take hours using ILP. Finally, the calculation is path-sensitive and introduces no pessimism as we shall demonstrate in Section 5.2.

3. Definitions

The core unit of the proposed calculation method is the basic block which is a contiguous sequence of instructions where the first instruction is jumped to (or is the first instruction of the program) and the last instruction is jumped from (or is the last instruction of the program) [11]. Each basic block B_i ($i \in [1..n]$) is associated an execution count x_i and an execution time c_i. When B_i has θ_i execution times, these will represented as c_i^j where $j \in [1..\theta_i]$. We also define $wcet_i$, $bcet_i$ to be the WCET and BCET of block B_i respectively.

The program under-analysis is represented by a control-flow graph (CFG) which is defined as a tuple (V, E). V is the set of vertices, in this case the basic blocks in the program, $|V| = n$. E is the set of edges, which in this case are the transitions between the basic blocks in the program.

In order to express conditional execution-time observation, we use the operator '$/$'. For example $c_{j/i} = \alpha$ means that $c_j = \alpha$ iff B_i is executed. A block B_i is the predecessor of a block B_j if there is sequence of one or more edges from B_i to B_j not containing a back-edge [11].

4. Deriving the Constraints

We use *predicated WCET analysis* which we define as performing WCET analysis by considering all different execution times of a program segment and expressing them as the outcomes of executing some other segments in the past. There is therefore the need to (a) identify the different execution times of a program segment and (b) identify the -previously executed- program segments that cause these execution times. Without loss of generality, we use the basic blocks as our program segments.

4.1. Constraints Using SA

The number of different execution times of a basic block varies with the complexity of the architecture where it runs. A complex architecture causes a basic block to exhibit a large number of different execution times. So far, we can express execution dependency constraints with respect to instruction caches (icaches), data caches (dcaches), static branch predictors, and pipelines. To maintain clarity, we will only address the derivation of icache constraints.

The analysis of the icache deals with deriving execution time dependencies between blocks in the CFG by exploiting basic-block layout in main memory[2]. We start by finding the WCET and BCET of all basic blocks. If B_i shares a program line with B_j, then the effect is expressed using the constraint $x_i > 0 \Rightarrow c_j = c_{j/i}$. We determine how many program lines belonging to B_j are loaded by the execution of B_i, let this be α lines. Block B_j acquires a new execution time $c_{j/i} := wcet_j - \alpha \times (i_m - i_h)$ where i_m is the icache miss latency, and i_h is the icache hit latency. If B_i displaces a program line used by B_j in a loop, then similarly $c_{j/i} := bcet_j + \beta \times (i_m - i_h)$.

The analysis is easily automated. For each block B_i, we find the blocks that share program lines with it, and the blocks that conflict with it in some icache blocks. The start and end addresses of the basic blocks, together with knowledge about the icache architecture enable the block execution-time dependency-analysis with respect to the icache.

[2]The CFG is constructed from the disassembled binary file of the program and hence basic block start and end addresses - in main memory - are available.

Studying cache conflicts is not novel in this work as it has been used in SA in the past [12]. The novelty here is in using SA to derive new execution times and link these execution times to past execution. Müeller [12] performed a complete analysis on instruction caches where icache accesses are identified as being hits, misses or unknowns. Since the analysis must be safe, unknowns are considered as being misses. This can be a great potential of pessimism in the evaluated WCET. In Figure 2(a) - ignoring the constraints - the returned WCET is 3200 assuming $c_3 = 70$.

4.2. Constraints Using Traces

Execution-trace analysis can also be used to derive the constraints of the CLP problem. An execution trace is a time-stamped execution of the program which can be obtained using a tracing method [13]. It contains all instructions executed during a particular run of the program with timing information. The execution trace can be exploited to derive constraints on the execution-counts of program segments or constraints on their execution-times.

Conditional execution times can be learnt from traces where a particular execution time of a block B_1 is recorded whenever B_2 is executed. The quality of the generated traces affects the correctness of the derived constraints. If traces are generated using full-path coverage, it is guaranteed that the timing constraints are learnt exactly. However, path coverage is impractical, so a less costly coverage metric must be employed. Unfortunately, functional testing coverage metrics are not adequate for our task as they do not consider the temporal properties of the program, and hence there is a need for new coverage metrics. We are currently exploring ways of generating appropriate test vectors that help obtain maximum variability in block execution times and executed paths using genetic algorithms.

5. Modeling the Constraints

In the last section we explain how conditional execution-times are expressed using implications. In this section we explain how CLP is used to model these constraints. In order to see the benefits of using CLP in our work, a comparison against ILP is made to illustrate the constraints that can be expressed better using CLP. In literature [9, 14], the constraints used in ILP are *flow* constraints. We use flow constraints and introduce *time constraints*.

5.1. Flow Constraints

These constraints express the rules governing the execution flow and dependencies in a program, these are divided into structural and functional constraints. Structural constraints preserve the execution flow of the program, and functional constraints describe aspects of program-execution behaviour. For a formal description, see [9, 14]. For instance, for any two blocks B_i and B_j, if we want to express that they are on the same path where B_j is inside a loop, the constraint $(x_i > 0 \land x_j > 0) \lor (x_i = 0 \land x_j = 0)$ is used in the CLP model. When the two blocks are outside any loop, the constraint $x_i = x_j$ is enough to express same-path relation. Mutual exclusion is represented similarly.

In this paper, the only type of flow constraints that is included in the CLP model are structural constraints. We do not detect functional constraints such as infeasible paths, so the model does not incorporate them. However, they can be added if available.

5.2. Time Constraints

These constraints describe the necessary conditions - expressed in terms of execution flow - that must hold to give rise to a particular execution time of some basic block. Given a basic block B_i with θ_i execution times $c_i^1, c_i^2, ..., c_i^\theta$, B_i is affected by a set Ψ_i of σ_i blocks $B_1, B_2, ..., B_{\sigma_i}$. In general $i \notin [1..\sigma_i]$, but in some special cases where the block size is larger than the icache size or when it accesses a variable whose size is larger than the dcache $i \in [1..\sigma_i]$. Every block B_k, $k \in [1..\sigma_i]$ can either execute or not execute, so there is a total of 2^{σ_i} different effects on block B_i. These effects are best visualized by imagining a truth table of θ_i variables where a 0 means block not executing and 1 means block executing. The relation $2^{\sigma_i} \geq \theta_i$ must hold because every execution time of a block B_i must be related to previous execution history. When $2^{\sigma_i} > \theta_i$, there will be some effects of the blocks in Ψ_i that are either equivalent (map to the same execution time) or impossible (the corresponding combination of blocks is not possible). Notice that in the architecture we consider, θ_i is usually small. Blocks B_i with large size can have a considerable θ_i.

Impossible effects can be ruled-out before passing the time constraints to the solver, or they can (eventually) be eliminated by the solver. Equivalent effects can be simplified using boolean algebra techniques and then passed to the solver. Obviously, the degree of simplification will be different for every equivalence class of time constraints.

Next we need to generate the conditional execution-time relations. Conditional execution times of B_i are expressed using

$$(x_1 \odot 0 \wedge x_2 \odot 0 \wedge ... \wedge x_{\sigma_i} \odot 0 \Rightarrow c_i = c_i^j) \tag{1}$$

where each $\odot$ stands for greater than ($>$) xor equal ($=$) (the $\odot$ can have a different instantiation in each occurrence in the same time constraint). The time c_i^j is the execution time observed for a given instantiation of the operators $\odot$ e.g. $(x_1 > 0 \wedge x_2 > 0 \wedge ... \wedge x_{\sigma_i} > 0 \Rightarrow c_i = 100)$.

Adding more constraints to the constraint model generally helps prune the search. The potential large number of time constraints is expected to speed-up the constraint search. For example, assume two blocks B_1, B_2 affecting the execution time of a third block B_3 in the following way:

$$\begin{aligned} &(x_1 = 0 \wedge x_2 = 0 \Rightarrow c_3 = 1) \\ &\wedge(x_1 = 0 \wedge x_2 > 0 \Rightarrow c_3 = 2) \\ &\wedge(x_1 > 0 \wedge x_2 = 0 \Rightarrow c_3 = 3) \\ &\wedge(x_1 > 0 \wedge x_2 > 0 \Rightarrow c_3 = 4) \end{aligned} \tag{2}$$

The search space is

$$(x_1, x_2, c_3) \in \{(\{0\}, \{0\}, \{1\}), (\{0\}, \mathbb{Z}^{+*}, \{2\}), (\mathbb{Z}^{+*}, \{0\}, \{3\}), (\mathbb{Z}^{+*}, \mathbb{Z}^{+*}, \{4\})\} \tag{3}$$

In the absence of these constraints, the search space is:

$$(x_1, x_2, c_3) = (\mathbb{Z}^+, \mathbb{Z}^+, \{1, 2, 3, 4\}) \tag{4}$$

The size of the search space in Formula 3 is $(|\mathbb{Z}^{+*}|^2 + 2 \times |\mathbb{Z}^{+*}| + 1)$. The size of the search space in Formula 4 is $(4 \times |\mathbb{Z}^+|^2)$. As can be seen, the time constraints partition the (non-linear) search space.

It is still possible to express conditional execution times in ILP at the cost of great complexity. This can be achieved through model duplication (ILP1) or bounds on execution times (ILP2).

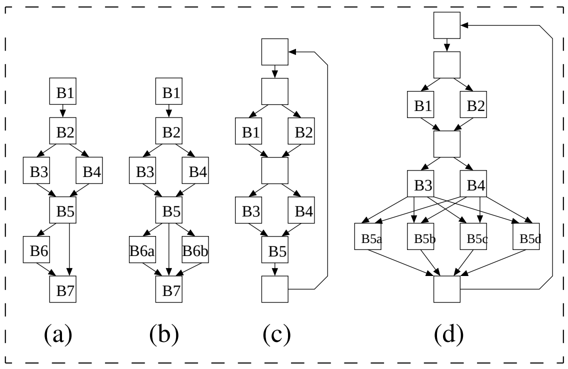


Figure 1: Time constraints for ILP

ILP1. Consider Figure 1(a) where $c_{6/3} = 7$ and $c_{6/4} = 10$. The basic ILP formulation of the problem when c_6 is constant is to maximize the sum $\sum_{i=1}^{7} c_i \times x_i$. When c_6 is not constant, the term $c_6 \times x_6$ needs to be expanded further. This is done by duplicating B_6 as is shown in Figure 1(b) and adding mutual-exclusive path information to the model.

In Figure 1(b), B_6 with execution times $\{7, 10\}$ is expanded to B_{6a} with execution time $c_{6a} = 7$ and B_{6b} with execution time $c_{6b} = 10$. Since $c_6 = c_{6a} = 7$ is observed only when B_3 is executed, we can state that B_{6a} is mutually exclusive with B_4. The same argument is made for B_{6b} and B_3. The updated ILP formulation becomes $(c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_{6a}x_{6a} + c_{6b}x_{6b} + c_7x_7)$ with the additional constraints that express mutual exclusivity. The ILP problem needs to be solved for each set of mutual exclusive paths, then the best solution is taken. The number of model copies to solve grows exponentially with the number of nodes that have multiple execution times and the number of different times they have (e.g. Figures 1(c), 1(d)).

ILP2. The other way to express conditional execution-times is to impose bounds on the number of times each single execution time is observed. This allows all constraints to be solved by a single run of the model and with expanding only the blocks in question. However, this only works provided the bounds on the observation of different execution times are available which is very hard to determine statically. In addition, the returned WCET will not be as accurate as the WCET returned by CLP or ILP1. The reason for this is that there is no path information in ILP2 compared to CLP, ILP1.

6. Example of Integration

Figure 3 shows (a) a program written in pseudo-assembly, (d) its CFG, (b) its block memory layout, and (c) the referenced variables placement in the dcache. We are interested in analysing the execution time of B_4 which has the value $wcet_4$ in its worst case. This is equivalent to performing 3 icache misses, 2 dcache misses, and starting execution from a flushed pipeline. Block B_4 is reached from three blocks: B_1, B_2, B_3 where no block in these three blocks is a predecessor of another one. Assume all blocks are outside any loop.

If B_4 is executed after B_1, it gains nothing in execution time with regards to icache because B_1 loads program lines PL_1 and PL_2 neither of which is used by B_4. Block B_4 however gains in dcache execution by $1 \times (d_m - d_h)$ because B_1 loads data line DL_1 which is used by B_4. Finally, B_4 gains

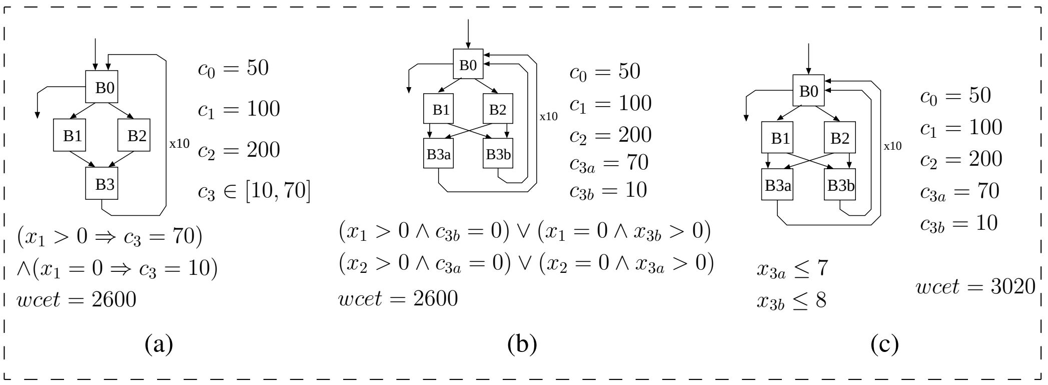

B0
B1
B2
B3
x10
$c_0 = 50$
$c_1 = 100$
$c_2 = 200$
$c_3 \in [10, 70]$
$(x_1 > 0 \Rightarrow c_3 = 70)$
$\wedge(x_1 = 0 \Rightarrow c_3 = 10)$
$wcet = 2600$
(a)
B3a
B3b
$c_{3a} = 70$
$c_{3b} = 10$
$(x_1 > 0 \wedge c_{3b} = 0) \vee (x_1 = 0 \wedge x_{3b} > 0)$
$(x_2 > 0 \wedge c_{3a} = 0) \vee (x_2 = 0 \wedge x_{3a} > 0)$
$wcet = 2600$
(b)
$x_{3a} \leq 7$
$x_{3b} \leq 8$
$wcet = 3020$
(c)

Figure 2: The time constraints and the corresponding WCET using CLP (a), ILP1(b), and ILP2(c). In (a) the nodes not duplicated, conditional execution times are added. In (b) and (c), the nodes with variable execution times are duplicated. In (b), mutual exclusion constraints are added. In (c), bounds on the execution counts of nodes with variable execution times are added.

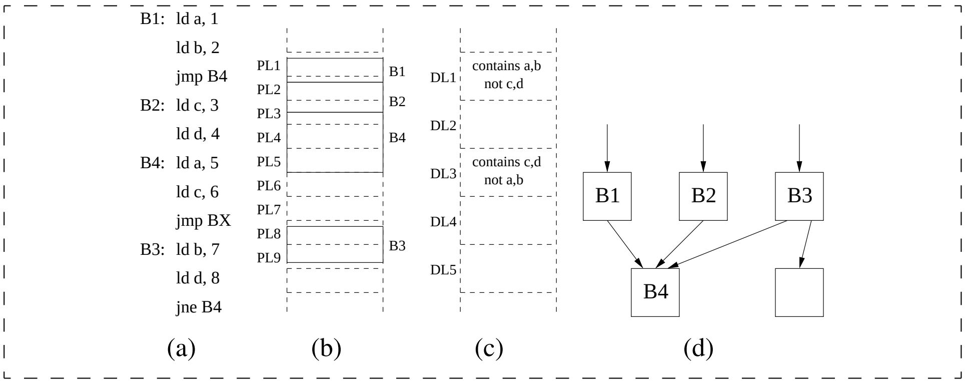


Figure 3: The disassembly code, memory layout, dcache content and a CFG window of a block B_4 affected by blocks B_1, B_2, and B_3

g_1 cycles because of the pipelined execution of $B_1; B_4$ (no misprediction can occur as the jump is unconditional).

If B_4 is executed after B_2, it gains $1 \times (i_m - i_h)$ with regards to icache because B_2 loads program line PL_3 which is used by B_4. Block B_4 gains in dcache execution by $1 \times (d_m - d_h)$ because B_2 loads data line DL_2 which is used by B_4. Finally, B_4 gains g_2 cycles because of the pipeline execution of $B_1; B_4$. Here assume $g_2 > g_1$.

If B_4 is executed after B_3, it gains nothing in execution time with regards to icache because B_1 loads no program line that is used by B_4. Block B_4 gains in dcache execution by $2 \times (d_m - d_h)$ because B_3 loads data lines DL_1, DL_2 which are used by B_4. Finally, B_4 gains g_3 cycles because of the pipeline execution of $B_3; B_4$. Here assume $g_2 > g_3 > g_1$.

The execution time of B_4 is captured by the constraint:

$$\begin{aligned}
&(x_1 > 0 \Rightarrow c_4 = wcet_4 - (d_m - d_h) - g_1)\wedge \\
&(x_2 > 0 \Rightarrow c_4 = wcet_4 - (i_m - i_h) - (d_m - d_h) - g_2)\wedge \\
&(x_3 > 0 \Rightarrow c_4 = wcet_4 - 2 \times (d_m - d_h) - g_3)
\end{aligned} \tag{5}$$

7. Evaluation and Results

We obtain the execution times using Simplescalar [3]. We use pollution techniques to force the WCET, BCET of each basic block in the program and compute this WCET, BCET by means of measurements. The hardware used comprises a CPU with a single-issue in-order pipeline, icache L1, dcache L1, and a static branch predictor. First, we analyse dependencies between basic blocks with respect to the icache as we discuss in Section 4.1 (the process is automatic). Then we derive the corresponding time constraints, and solve the constraints using ECL^iPS^e, a constraint logic programming engine [2].

The objective of the evaluation is to show (a) that PWA yields tighter WCET estimations in comparison with HMU, and (b) show that the solution time to solve the CLP model is affordable.

We compare the tightness of the WCET values obtained using our Predicated WCET Analysis (PWA) with a method that uses Hit, Miss, first-hit, first-miss, Unknown analysis (HMU) [12]. An HMU icache analysis method quantifies the number of icache hits and misses per basic block. When the icache access is guaranteed to be a hit or a miss, it is classified accordingly. When the icache access is not guaranteed to be a hit or a miss (i.e. unknown), it is classified as a miss to achieve safety. Our analysis method puts more context-sensitivity in the icache analysis by stating the condition under which the icache access (considered a miss by HMU) will hit or miss.

We have tested our tool on some WCET benchmarks available from [15]. Table 1 shows the execution times obtained using PWA and HMU on a representative[3] subset of the benchmarks. The icache has a size of $1k$ bytes. As can be seen, considerable tightness has been achieved in WCET for the first three programs (*select, fdct, fir*) which can be explained by the large number of constraints -relative to the number of blocks- which allows less pessimism during the constraint search.

The fourth program (*lms*) -although having the largest relative number of constraints in the table- does not have the best WCET tightness using PWA. This is due to the nature of the constraints involved in calculation. In our implementation, as a temporary solution to manage the large number of constraints that a particular block can have, we decide that each basic block can be constrained by at most five blocks. When a block is constrained by more than five blocks, its WCET is used during calculation. The program *lms* has got a big loop which consumes more than half the number of its blocks (73), which leads to many icache conflicts given the used icache configuration.

The last three programs (*cnt, bsort,* and *ns*) scored very small tightness. When these programs are run on an icache with smaller size, they generate more constraints and score greater tightness.

The CLP solving time during evaluation (including some other programs) did not exceed a few seconds. The solving process in ILP is usually instantaneous for one run but then becomes exponential

[3]Representative in terms of tightness i.e. the tightness scored with other programs from the benchmarks has more or less one of the values shown in Table 1.

Table 1: WCETs of benchmark programs using PWA and HMU

#	program	blocks	implications	wcet		gain
				HMU	PWA	
1	select	40	27	558627	432803	22.6%
2	fdct	12	6	77759	66975	15%
3	fir	17	4	87822	81742	7%
4	lms	134	86	747776	724752	4.3%
5	cnt	36	2	94672	92912	1.9%
6	bsort	20	4	58179	57539	1.2%
7	ns	22	5	892708	888148	0.6%

when running all duplications. We use *lp_solve* to solve each of the (linear) disjunctive ILP instances (ILP1). Each instance is solved in few micro seconds. Using a more powerful ILP solver such as *CPLEX* might cut down the time required to solve one instance of the disjunctive ILP. However, this will only mean that (few) more time constraints can be tolerated. The exponential behaviour is still present.

If for instance, the model has n time constraints and *lp_solve* takes α units to solve each instance of ILP1; the number n' of time constraints that can be solved using *CPLEX* in the same amount of time is $n' = n - ln(2) \times ln(\alpha/\beta)$ where β is the time taken by *CPLEX* to solve one instance. So if *CPLEX* was a million times faster than *lp_solve* ($\alpha = 10^6 \times \beta$), *CPLEX* can solve the same model with extra 10 time constraints in the same amount of time. The models in our case were solved in few micro seconds, if the solver was a million times faster or more, they would be solved in few pico seconds or less which is doable only by super computers.

8. Conclusions and Future Work

In this paper we have proposed the use of constraint-logic programming (CLP) to compute tight values of WCET by using constraints derived through execution-time dependency-analysis. In this work we have considered icache constraints only and we concluded that CLP is superior to integer linear-programming (ILP) whenever there is a reasonable number of execution-time dependencies. The choice of whether or not to use predicated WCET analysis (PWA) and CLP is dictated by the nature of the program. If execution-time dependency-analysis reveals lots of constraints, it is worth using PWA and CLP because considerable tightness may be achieved. In a future work, we will show how constraints from other hardware components are derived. We are currently investigating how to prove the safety of constraints derived using tracing. We are also working on improving the calculation method so that constraints are solved more efficiently.

References

[1] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[2] K.R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse.* Cambridge University Press, New York, NY, USA, 2007.

[3] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

[4] C. Burguiere and C. Rochange. A contribution to branch prediction modeling in WCET analysis. In *Proceed. of the conf. on Design, Automation and Test in Europe*, pages 612–617, Washington, USA, 2005. IEEE Computer Society.

[5] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, April 2000.

[6] J.F Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on Worst Case Execution Time Analysis*, pages 13–16, Palma de Mallorca, Spain, July 2005.

[7] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, August 2003.

[8] C.A. Healy, R.D. Arnold, F. Müeller, M.G. Harmon, and D.B. Walley. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, 1999.

[9] Y.T. Steven Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[10] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. In *Proceedings of the 2nd Workshop on WCET Analysis*, October 2002.

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

[12] F. Müeller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, 2000.

[13] Stefan M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *Proceedings of the 3rd International workshop on worst-case execution time (WCET) analysis*, pages 75–78, July 2003.

[14] P. Puschner and A.V. Schedl. Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[15] Mälardalen WCET research group. Wcet project/benchmarks. *http://www.mrtc.mdh.se/ projects/wcet/benchmarks.html*, January 2008.

[16] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3):109–128, 2007.

TRACES AS A SOLUTION TO PESSIMISM AND MODELING COSTS IN WCET ANALYSIS

Jack Whitham and Neil Audsley[1]

Abstract
WCET analysis models for superscalar out-of-order CPUs generally need to be pessimistic in order to account for a wide range of possible dynamic behavior. CPU hardware modifications could be used to constrain operations to known execution paths called traces, permitting exploitation of instruction level parallelism with guaranteed timing. Previous implementations of traces have used microcode to constrain operations, but other possibilities exist. A new implementation strategy (virtual traces) is introduced here.

In this paper the benefits and costs of traces are discussed. Advantages of traces include a reduction in pessimism in WCET analysis, with the need to accurately model CPU internals removed. Disadvantages of traces include a reduction of peak throughput of the CPU, a need for deterministic memory and a potential increase in the complexity of WCET models.

1. Introduction

Worst-case execution time (WCET) analysis determines the maximum execution time for a program on a particular CPU [17]. On some CPUs, the execution of each instruction within the program is independent of execution history and data inputs, making the program easy to model for WCET analysis. Methods like the *implicit path enumeration technique* (IPET) [10] can find the exact WCET in these simple cases because the execution time of each section of code can be considered independently of all the others [18]. If the execution times are not constant, then it is still possible to use an upper bound on the execution time of blocks of code, yielding a *pessimistic* (i.e. inexact but safe) WCET estimate. In IPET, the program is modeled as a flow network, and *integer linear programming* is used to determine the execution path with the maximal execution time.

Out-of-order CPUs are difficult to analyze due to their complexity, but accurate analysis is necessary because a safe upper bound is required [8]. Advanced CPUs provide higher instruction throughput by using very long pipelines, executing operations in parallel, and executing operations *speculatively* (that is, before the branch that led to them): all of these features must be modeled. The rate of code execution is dependent on history (via branch prediction [5], caches [13] and pipeline interaction effects [11]) and dependent on data values (because of memory disambiguation [12] and variable duration instructions). Consequently, the upper bound may be much greater than the typical case. In some cases, a slower and simpler CPU might have a lower WCET as well as facilitating analysis.

Previous work suggested the use of microcoded *traces* as an abstraction of CPU behavior for WCET analysis [23, 22]. Traces support speculative and parallel execution within an IPET-based WCET

[1]Real-Time Systems Group, Department of Computer Science, University of York, York, YO10 5DD, UK.
email: {jack/neil}@cs.york.ac.uk

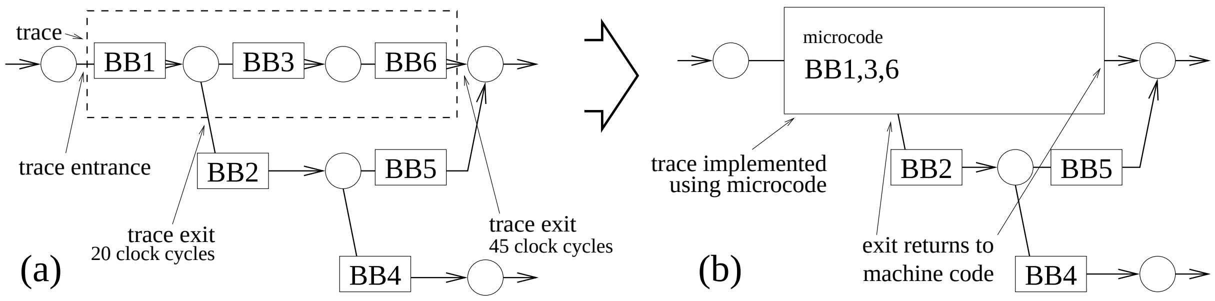


Figure 1. Trace formation in [23]: (a) basic blocks on the WCEP are selected for inclusion in a trace, then (b) their instructions are rescheduled into blocks of microcode, reducing the execution time of the path.

analysis model, potentially *without* introducing pessimism. Each trace is a sequence of basic blocks: a subpath of a program. Trace allocation algorithms try to match traces to subpaths of the *worst-case execution path* (WCEP) which maximizes a program's execution time. The WCEP might be changed by each trace allocation, so it is (1) initially estimated by assuming fixed execution times for code and performing WCET analysis, and then (2) refined by repeating WCET analysis after N trace allocations. When a path has been selected for trace formation (Figure 1(a)), a scheduler converts the basic blocks into functionally equivalent microcode (Figure 1(b)). The microcode is explicitly parallel and uses speculation to execute the operations along the main path [BB1, BB3, BB6] as quickly as possible. Hence, the local WCET of that path is reduced, and the WCET of the whole program may also be reduced if the main path is chosen correctly. The cost of adding the trace is that other paths may have increased execution time (e.g. [BB1, BB2]), so allocation algorithms must evaluate the overall WCET reduction benefits of each choice.

This paper details the motivation for traces using previous work (section 2) then describes the timing model that they enable (section 3). Then a new implementation strategy is discussed, avoiding microcode (section 4). Section 5 concludes.

2. Why use Traces?

Out-of-order CPUs exploit instruction level parallelism using a complex heuristic mechanism that attempts to run operations as soon as possible. The incoming instructions specify code for a sequential machine: an out-of-order CPU preserves in-order semantics while executing code in parallel where possible. The execution rate is limited by the data dependences in the code, the accuracy of predictions about future control flow, the resource limits of the CPU and the memory bandwidth. The first out-of-order pipelines were implemented in the 1960s using two different mechanisms to track dependences: a *scoreboard* (in the CDC 6600 [2]) and Tomasulo's algorithm (in the System 360 CPU [20]). Since then, the designs have been greatly refined [19].

As part of real-time systems design, it is necessary to compute the WCET of various programs running on the CPU. If the CPU is an out-of-order CPU, then building an accurate model will be costly [8]. The costs can be reduced by improving CPU predictability, e.g. by using locked caches [6] or scratch-pads [15] to replace unpredictable memory systems, and replacing dynamic branch predictions with static predictions [4]. But dynamic behavior still exists in the operation scheduler which is affected by history and data dependences. Accounting for all possible behaviors turns WCET computations into pessimistic estimates, so the CPU resources are under-utilized, particularly if a suboptimal condition such as a *domino effect* [11] or *timing anomaly* [21] is possible. Such conditions are handled by incorporating pessimism into the CPU model [9] or by adding pipeline synchronizing instructions to the code to prevent the effect [11]: both approaches increase the WCET estimate.

To *avoid* CPU modeling, the probabilistic WCET approach has been proposed, where a statistical model of the execution time of a program is built automatically using measurements [3], but this approach is not suitable for all applications because the upper bound cannot be guaranteed. Alternatively, modeling costs can be *reduced* by constraining a complex CPU to intermediate deadlines obtained using a simpler CPU model. In VISA [1], programs execute on an out-of-order CPU that is downgraded to predictable in-order operation if an intermediate deadline is not reached. Unfortunately, this limits the WCET to that of an in-order CPU. Finally, modeling costs can be eliminated entirely by *single path programming* [16] where branches are replaced by predicated execution, since single path programs have constant execution times. Here, the WCET is limited by predication effects, since instructions in both the *if* and *else* cases of a conditional statement pass through the CPU.

The trace approach is related to all of the above. As in [3], it is observed that superscalar out-of-order CPUs are (1) difficult to model, and (2) models are pessimistic in any case, so it is best to avoid making a model of the CPU. Measurements obtained for probabilistic WCET allow statistically valid observations to be made regarding the WCET, but traces require a finite number of measurements which always cover all possible behaviors while probabilistic WCET requires a potentially unbounded number of measurements for a high degree of confidence in the WCET computation. As in [16], it is observed that removing control flow permits direct measurement. Traces remove the need to model internal control flow within the CPU (leaving program control flow) while single path programming removes both types of control flow. Finally, as in [1], an out-of-order CPU is modified to provide guarantees about timing, but the trace model accommodates speculative out-of-order execution instead of enforcing an upper timing bound.

3. What is a Trace?

In this paper, a trace is (1) executable code and (2) a timing model to represent the properties of that code. As executable code, a trace replaces sequential machine code in one or more basic blocks, forming part of a path through the program. It is *functionally equivalent* to that code, but *executes in less time*, at least along its *main path*. This is the same definition that is widely used in previous work, e.g. [7]. Traces have been previously implemented using microcode [23], but this is not necessary (section 4). An entire program could be composed of traces (as in this paper), or traces might be combined with predictable in-order execution (as in [23]). As a timing model, a trace is a subgraph of a *timing graph* (T-graph), as proposed in [18] for WCET analysis using IPET. The model is:

1. A trace always begins execution with the internal parts of the CPU in a well-defined state. The next instruction to be executed is the beginning of a basic block e, known as the *entrance*.

2. A trace has $1 \leq n \leq L + 1$ exits: when these are reached, execution may move to another trace. Figure 2(a) shows a trace with three exits.

3. A trace requires a precisely known number of clock cycles to reach each one of the n exits from the entrance. The path to exit i from entrance e is denoted as $P_{e,i}$ for WCET analysis purposes: $P_{e,i}$ is a sequence of basic blocks. The time taken is $t(P_{e,i})$.

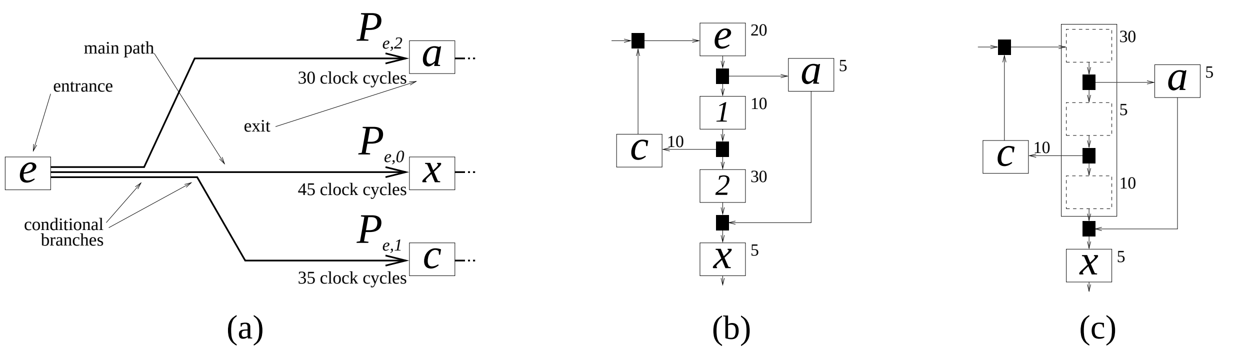


Figure 2. (a): Three paths through a trace beginning at basic block e. The paths lead to basic blocks a, x and c. (b): T-graph containing a, x and c. (c): T-graph incorporating the trace. The execution cost of each basic block is shown. The cost of the paths $e \rightarrow a$ and $e \rightarrow c$ increases slightly, but the cost of the main path $e \rightarrow x$ is decreased.

4. A trace contains up to L conditional branches along the *main path* $P_{e,0}$ ($e \rightarrow x$ in Figure 2(a)). Every other path $P_{e,j}$ ($j \neq 0$) also follows this path until conditional branch j is reached. Then, $P_{e,j}$ leads to an exit while $P_{e,0}$ continues.

5. An exit is taken when a branch condition is evaluated as True or the main path's end is reached.

6. After any exit, a transformation has been applied to the program state (i.e. general-purpose registers, program counter and RAM). The transformation is guaranteed to be identical to the transformation that would have been applied if the original machine code had been executed.

The purpose of the trace is to reduce the execution cost of the main path $P_{e,0}$ by permitting speculation and out-of-order execution along this path. The cost of other paths may be reduced or increased. Because each $P_{e,i}$ is constant, it is possible to use exact IPET analysis (without pessimism): every trace is composed of "basic blocks" in microcode, each with constant execution times, permitting IPET to determine exact results [18]. The T-graph shown in Figure 2(b) is transformed to the T-graph in Figure 2(c) by the trace shown in Figure 2(a). More complex transformations are required when a trace represents an unrolled loop, because a basic block may be executed in multiple contexts [22].

4. Constraining CPU Behavior

The dynamic operation scheduler's behavior can be predicted precisely if hardware exists to (1) reset the scheduler to a known state, and (2) constrain all of the external inputs that could affect it. This can be used to implement *virtual traces*, which share the trace timing model (section 3) but use the dynamic operation scheduler in place of microcode. Figure 3 shows a diagram of a dynamic scheduler with external inputs, showing every source of *noise* that could affect execution. To fit the timing model in section 3, virtual traces must specify a *main path* through the program, and the execution time of that path (and all exit paths) must be an exact number of clock cycles. To implement virtual traces, dynamic scheduler inputs are restricted as follows:

• *Cache stalls* can be eliminated by cache locking [6] or by using *scratchpad memory* [15]. Like caches, scratchpads are on-chip memories that can allow programs to avoid slow and energy-intensive accesses to off-chip memory. But unlike caches, scratchpads are not automatically updated during program execution: they must be explicitly loaded by a program [14]. This is not as convenient as a

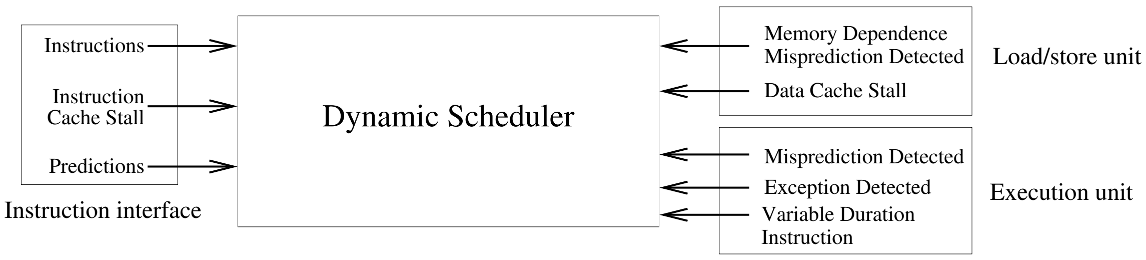


Figure 3. Sources of noise that could affect the operation of a dynamic operation scheduler.

cache but it is easy to predict the latency of a memory access. No cache modeling [13] is required, so the complexity of analysis is reduced.

• *Memory dependence mispredictions* [12] can be eliminated by enforcing a safe ordering on memory operations: load operations cannot be reordered across store operations, and store operations cannot be reordered at all. Load/store forwarding is disabled as it is data dependent.

• *Variable duration instructions* can be eliminated by forcing a fixed (upper bound) duration.

• *Exceptions* are discarded; many programs do not use them. (They could be modeled as conditional branches if necessary.)

Other inputs shown in Figure 3 are accommodated:

• *Branch predictions* fit into the trace model; the dynamic branch predictor is replaced by a representation of the trace. It (1) generates predictions so that instruction fetching follows the main path through the trace, and (2) considers the detection of a misprediction as an exit from the trace. However, the hardware must ensure that *branch operations are executed in program order*, since that prevents $n > 1$ misprediction events being active at the same time, leading to up to $2^n - 1$ possible exit conditions instead of 1. This can be done through the instruction dependence mechanism.

• *Instructions* also fit into the trace model: they are fetched along the main path, and when the end of the main path is reached, fetching is stalled. This prevents further instructions introducing noise.

The previous state of the scheduler may also have an effect on the schedule. This could be handled by (1) adding a reset function or (2) stalling the incoming instructions until the pipeline is drained.

4.1. Benefits

The CPU modifications guarantee that the operation scheduler is not affected by execution history (except within each trace) and that operation is not data dependent. This allows speculation and out-of-order execution along the main path. The speculation that occurs is always predictable. Inputs always arrive at known intervals, so the scheduler always does the same thing: following one of the paths $P_{e,i}$ through each trace.

The changes affect the load/store unit (removal of load/store forwarding) and the execution unit (removal of variable duration instructions). There are new dependences for branches and memory operations. Finally, a device is added to manage the execution of virtual traces (Figure 4). This is a

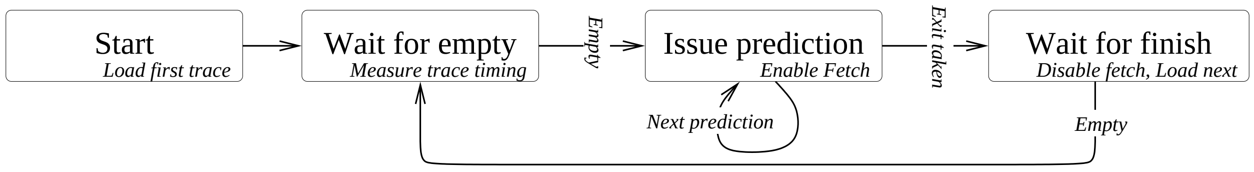


Figure 4. Virtual trace controller state machine: waits for the pipeline to empty before beginning the next trace.

simple state machine that enforces a strict order on trace execution, ensuring that each trace has fully completed before the next one begins.

This arrangement allows each $t(P_{e,i})$ value to be measured using the CPU, which is treated as a black box. Hence, CPU modeling costs are very low. Given these timing values, programs composed of traces can be represented within an IPET model as in [23]. The only limit on the number of traces is the storage space required: for virtual traces, the stored data comprises branch predictions and the length, so space requirements are minimal. As in VISA [1], the CPU modifications could be turned on and off dynamically, allowing real-time tasks to be mixed with non-real-time tasks on the same platform without any interference between the two. The approach could also be combined with single path programming [16] by using predication to remove conditional branches in frequently executed code: this could be beneficial since fewer exits would exist in each trace, and consequently opportunities for parallelism would be increased.

4.2. Costs

Previous work suggests three potential problems with traces, independent of the implementation:

• *Reduction in Peak Throughput* - this is almost certain to be lower than the peak throughput of a similarly-configured CPU optimized for ACET reduction. For example, the pipeline is emptied at every exit from a trace. Typical CPU designs would attempt to do useful work during this time, such as executing the next piece of code, but that could interfere with subsequent timing. Enforcing an order on memory operations will also reduce throughput [12].

• *Deterministic Memory Assumption* - every memory access must respond in a known time period. Cache stalls disturb the operation of the pipeline, perhaps introducing timing anomalies [21]. In an environment with multiple CPU cores, this could be particularly problematic as bus contention would also be a factor. Scratchpads could be used as a replacement for caches, but this increases the engineering difficulty of building the program [15, 14].

• *Analytical Complexity* - the IPET model becomes more complex when traces are introduced, because (1) there are more basic blocks, and (2) the new trace basic blocks are linked to the constraints on the original basic blocks [23]. Although the total number of integer linear program constraints is only increased by $O(n)$ for a program with n basic blocks, the difficulty of solving the IPET problem could still be vastly increased due to the NP-hard nature of integer linear programming problems.

5. Conclusion

This paper has explained the motivation for traces, outlined a WCET analysis model for them, and described a way to implement virtual traces by modifying a superscalar out-of-order CPU. Likely benefits and costs have been discussed.

Future work will use a simulated implementation of virtual traces to determine the exact costs of the restrictions on throughput. It would be interesting to compare these to the pessimistic assumptions that would otherwise need to be made in order to determine the WCET. Intuitively, the pessimism inherent in traces is likely to be lower than the pessimism from analysis, and consequently traces could provide higher guaranteed performance. Low CPU modeling costs are another benefit. Despite this, the disadvantages of traces (section 4.2) may be prohibitive. Further research will provide more information about the costs and benefits of the ideas.

6. Acknowledgments

Thanks to the anonymous reviewers for their helpful suggestions.

References

[1] Aravindh Anantaraman, Kiran Seth, Eric Rotenberg, and Frank Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA). In *Proc. RTSS*, pages 114–125, 2004.

[2] Gordon Bell. CDC 6600 registers (online, accessed 3 June 08). `http://research.microsoft.com/~gbell/craytalk/sld040.htm`.

[3] Guillem Bernat, Alan Burns, and Martin Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Comput.*, 1(2):179–194, 2005.

[4] Francois Bodin and Isabelle Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proc. ECRTS*, pages 33–40, 2005.

[5] Claire Burguiere and Christine Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Proc. DATE*, pages 612–617, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.

[7] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.

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

[9] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. DAC*, pages 456–461, 1995.

[11] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12, 1999.

[12] Andreas Moshovos. Exploiting load/store parallelism via memory dependence prediction. In *Speculative Execution in High Performance Computer Architectures*, pages 355–392. CRC Press, 2005.

[13] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000.

[14] Isabelle Puaut and Damien Hardy. Predictable paging in real-time systems: A compiler approach. In *Proc. ECRTS*, pages 169–178, 2007.

[15] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, 2007.

[16] Peter Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. ECRTS*, Technical Report, Jun. 2002.

[17] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[18] Peter Puschner and Anton Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Syst.*, 13(1):67–91, 1997.

[19] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359.

[20] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. R&D*, 11(1):25–33, 1967.

[21] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[22] Jack Whitham. Real-time processor architectures for worst case execution time reduction. PhD Thesis YCST-2008-01, University of York, 2008.

[23] Jack Whitham and Neil Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proc. RTAS*, pages 305–316, 2008.

TUBOUND –A CONCEPTUALLY NEW TOOL FOR WORST-CASE EXECUTION TIME ANALYSIS [1]

Adrian Prantl,[2] Markus Schordan[2] and Jens Knoop[2]

Abstract

TUBOUND *is a conceptually new tool for the worst-case execution time (WCET) analysis of programs. A distinctive feature of* TUBOUND *is the seamless integration of a WCET analysis component and of a compiler in a uniform tool.* TUBOUND *enables the programmer to provide hints improving the precision of the WCET computation on the high-level program source code, while preserving the advantages of using an optimizing compiler and the accuracy of a WCET analysis performed on the low-level machine code. This way,* TUBOUND *ideally serves the needs of both the programmer and the WCET analysis by providing them the interface on the very abstraction level that is most appropriate and convenient to them.*

In this paper we present the system architecture of TUBOUND*, discuss the internal work-flow of the tool, and report on first measurements using benchmarks from Mälardalen University.* TUBOUND *took also part in the WCET Tool Challenge 2008.*

1. Motivation

Static WCET analysis is typically implemented by the implicit path enumeration technique (IPET) [15, 19] which works by searching for the longest path in the *interprocedural control flow graph (ICFG)*. This search space is described by a set of *flow constraints* (also called flow facts), which include e.g. upper bounds for loops and relative frequencies of branches. Flow constraints can generally be determined by statically analyzing the program. However, there are many cases where a tool has to rely on *annotations* that are provided by the programmer, because of the undecidability of certain analysis problems or imprecision of the analyses. Current WCET analysis tools, as they are used by the industry, therefore allow the user to annotate the machine code with flow constraints.

The goal of the TuBound approach is to lift the level of user annotations from machine code to source code, while still performing WCET analysis on the machine code level. In addition to keeping the precision of low-level WCET analysis, this has the following benefits:

- *Convenience and Ease*: For the user, annotating the source code is generally easier and less demanding as annotating the assembler output of the compiler.
- *Reuse and Portability*: Source code annotations, which specify hardware-independent behaviour, can directly be reused when the program is ported to another target hardware.

[1]This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis*, `http://costa.tuwien.ac.at/`, the ARTIST2 Network of Excellence, `http://www.artist-embedded.org/`, and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

[2]Institut für Computersprachen, Vienna University of Technology, Austria
email: {adrian,markus,knoop}@complang.tuwien.ac.at

- *Feedback and Tuning*: Source code annotations can be used to present the results of static analyses to the programmer for inspection and further manual refinement.

The major obstacle, which has to be overcome for realizing such an approach, is imposed by the fact that compiler optimizations can modify the control-flow of a program and thus invalidate source code annotations. In TUBOUND, this is taken care of by transforming flow constraints according to the performed optimizations. Technically, this is achieved by a special component, called FLOWTRANS, which is a core component of TUBOUND and described in Section 3.2. FLOWTRANS performs source-to-source transformations. Therefore, our overall approach is retargetable to other WCET tools; currently we are using CALCWCET$_{167}$.

From the tool developer's point of view, this source-based approach offers the advantage that analyses can use high-level information that is present in the source code, but would be lost during the lowering to an intermediate representation. A typical example for such information is the differentiation between bounded array accesses and unbounded pointer dereference operations. Since the output of a source-based analysis is again annotated source code, it is also possible to create a feedback loop where the user can run the static analysis and fill in the annotations where the analysis failed to produce satisfying results. Afterwards, the analysis could be rerun with the enriched annotations to produce even tighter estimates.

TUBOUND is based on earlier work by Kirner [14] who formulates the correct flow constraint updates for common compiler transformations. TUBOUND goes beyond this approach by extending it to source-to-source transformations and by adding interprocedural analysis. Optimization traces for flow constraint transformations are also used by Engblom et al. [8]. With FLOWTRANS, we are taking this concept to a higher level, by performing control-flow altering transformations already at the source code level. Another approach towards implementing flow constraint transformation was recently described by Schulte [22]. In contrast to TUBOUND, this approach is based on the low-level intermediate representation of the compiler. The integration of static flow analysis and low-level WCET analysis is also implemented in the context of SWEET, which uses a technique called abstract execution to analyse loop bounds [9, 10]. Again, our approach uses a higher level of abstraction by performing static analyses directly at the source code level. The interaction of compiler optimizations and the WCET of a program has been covered by Zhao et al. [24], where feedback from a WCET analysis was used to optimize the worst-case paths of a program.

2. The architecture of TuBound

TUBOUND is created by integrating several components that were developed independently of each other. The majority of the components is designed to operate on the source code. This decision was motivated by gains in flexibility for both tool developer and users.

The architecture and work flow of TUBOUND is summarized in Figure 1. The connecting glue between the components is the *Static Analysis Tool Integration Engine* (SATIrE) [20, 6]. SATIrE enables using data flow analyzers specified with the *Program Analyzer Generator* (PAG) [16, 3] together with the C++ infrastructure of the ROSE compiler [21]. SATIrE internally transforms programs into its own intermediate representation, which is based on an abstract syntax tree (AST). An external term representation of the AST can be exported and read by SATIrE. This term representation is generated by a traversal of the AST and contains all information that is necessary to correctly unparse the program. This information is very fine-grained and includes even line and column information of the respective expressions. The terms are also annotated with the results of any preceding static analy-

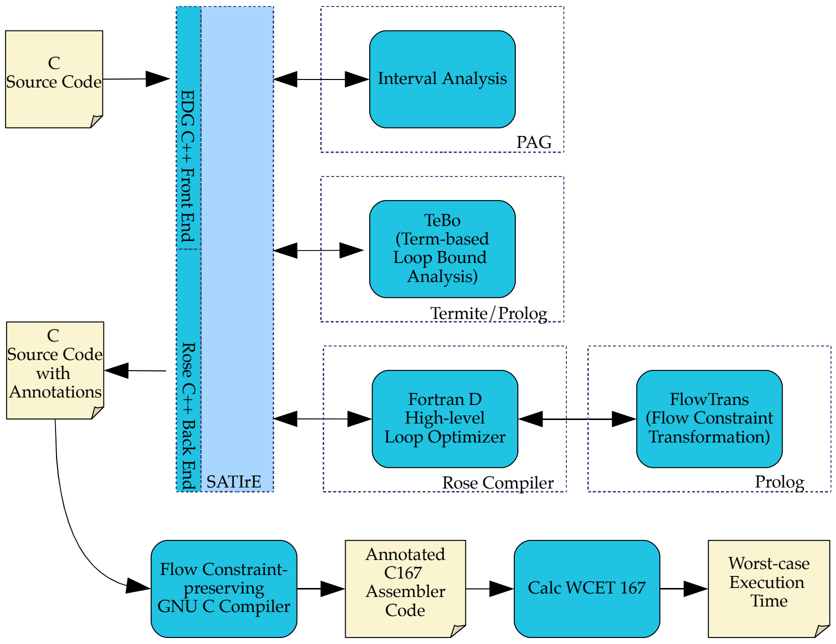

C Source Code
EDG C++ Front End
Rose C++ Back End
SATIrE
Interval Analysis
PAG
TeBo (Term-based Loop Bound Analysis)
Termite/Prolog
C Source Code with Annotations
Fortran D High-level Loop Optimizer
Rose Compiler
FlowTrans (Flow Constraint Transformation)
Prolog
Flow Constraint-preserving GNU C Compiler
Annotated C167 Assembler Code
Calc WCET 167
Worst-case Execution Time

Figure 1. The collaboration of TuBound's components

C source code	Term representation
`7   for (i = 0; i < 100; i++) {`	see below

```
for_statement(
  for_init_statement( [ expr_statement( assign_op(
    var_ref_exp(
      var_ref_exp_annotation(type_int,"i",0,
                                null,analysis_result(null,null)),
      file_info("triang.c",7,10)),
    int_val(null,value_annotation(0,analysis_result(null,null)),
            file_info("triang.c", 7, 12)),
    ... ], default_annotation(null, analysis_result(null,null)),
                 file_info("triang.c", 7, 3)),
    expr_statement( less_than_op(
      var_ref_exp(var_ref_exp_annotation(type_int,"i",0,null,
        ...
```

Figure 2. The external AST term representation of SATIrE

sis. The key feature, however, is the syntax of the term representation. It was designed to match the syntax of Prolog terms. A Prolog program can thus access and manipulate these terms very easily. A similar approach of using Prolog terms to represent the AST of a program is used in the JTransformer framework for the Java language [2].

The Rose compiler is a source-to-source transformation framework that includes the EDG C++ front end, a loop optimizer and a C++ unparser [21, 5]. The loop optimizer was ported from the Fortran D compiler. In TuBound we are using the front end and the high-level loop optimizer that is part of Rose. The Program Analyzer Generator (PAG) by AbsInt Angewandte Informatik GmbH allows the specification of data flow analyses in a specialised functional language [16, 3]. Using PAG, we implemented a variable interval analysis for TuBound. CalcWCET$_{167}$is a tool that performs WCET analysis for the Infineon C167 micro-controller [4]. CalcWCET$_{167}$ expects annotated C167 assembler code as input. The tool is complemented by a customized version of the GNU C compiler that translates annotated C sources into annotated assembler code for the C167 micro-controller.

3. The Work Flow of TuBound

Conceptually, the work flow of analysing a program with TuBound comprises three stages:

3.1. Start-up and Annotation

Parsing. In the first phase, the source code of the program is parsed by the EDG C++ front end that is integrated into the Rose compiler. Rose then creates a C++ data structure of the AST and performs consistency checks to verify its integrity. The Rose loop optimizer performs analysis and transformations based on the AST data structure.

Interval Analysis. The AST is traversed by SATIrE to generate the interprocedural control flow graph (ICFG), an amalgam of call graph and *intra*procedural CFG [23]. This data structure is the interface for the PAG-based interval analysis that calculates the possible variable value ranges at all program locations. The context-sensitive interval analysis operates on a normalized representation of the source code that is generated during the creation of the ICFG. The interval analysis is formulated as an interprocedural data-flow problem and is a pre-process of the loop bounding algorithm, which is otherwise unable to analyze iteration counts that depend on variable values that stem from different calling contexts. Once the interval analysis converges to a fixed point, the results are mapped back to the AST.

Loop Bound Analysis. The next step is the loop bound analysis. This analysis operates on the external term representation of SATIrE. We exploit this fact with our term-based loop bounder (TeBo) which was written entirely in Prolog. Our loop bounding algorithm exploits several features of Prolog: To calculate loop bounds, a symbolic equation is constructed, which is then solved by a set of rules. It is thus possible for identical variables with unknown, but constant values to cancel each other out. For example, in the code `for (p = buf; p < buf+8; p++)`, the symbolic equation would be $lb = (buf + 8 - buf)/1$. The right-hand side expression can then be reduced by TeBo's term rewriting rules. The loop bounding algorithm also ensures that the iteration variable is not modified inside the loop body. This is implemented with a control flow-insensitive analysis [17] that ensures that the iteration variable does not occur at the left-hand side of an expression inside the loop body and its address is never referenced within its scope.

In the case of nested loops with non-quadratic iteration spaces, loop bounds alone would lead to an unnecessary overestimation of the WCET. In TeBo, we are using constraint logic programming to yield generalized flow constraints that describe the iteration space more accurately. An example is shown in Figure 3. The nested loop in the example has a triangular iteration space, where the innermost basic block is executed $n * \frac{n-1}{2}$ times. Our analyzer finds the following equation system for this loop nest:

$$\begin{array}{lll} m3 & = \sum_{n=0}^{99} m3_n(\{i := n\}) & (1) \\ m3_n(env) & = n = i & (2) \\ m2 & = m1 * 100 & (3) \end{array}$$

The equations are constructed with the help of an *environment* that consists of the assignments of variables at the current iteration. The variable $m1$ stands for the execution count of the `main()` function, $m2$ for the count of the outer loop and $m3$ for the count of the innermost loop. Equation 1 describes the fact that the values of i as well as the iteration counts for the individual runs of the inner loop are $0..99$, respectively. Equation 2 describes the generic behaviour of the inner loop, stating that its iteration count is equal to the value of n in the current environment. The last equation describes the behaviour of the outer loop. The use of constraint logic programming allows for a lightweight

Original program

```
int main()
{
  int i,j;
  for (i = 0; i < 100; i++) {
    for (j = 0; j < i; j++) {
      // body
    }
  }
}
```

Annotations generated by TuBound

```
int main() {
#pragma wcet_marker(m1)
  int i;
  int j;
  for (i = 0; i < 100; i++) {
#pragma wcet_constraint(m2=<m1*100)
#pragma wcet_marker(m2)
#pragma wcet_loopbound(100)
    for (j = 0; j < i; j++) {
#pragma wcet_constraint(m3=<m_1*4950)
#pragma wcet_marker(m3)
#pragma wcet_loopbound(99)
      // body
    }
  }
  return 0;
}
```

Figure 3. Finding flow constraints with constraint logic programming

implementation that does not rely on additional tools. In earlier work, Healy et al. [11] are using analysis data to feed an external symbolic algebra system that solves the equation systems for loop bounds.

Eventually, the results of the loop bound analysis are inserted into the term representation as annotations of the source code. We are using the `#pragma` directive to attach annotations to basic blocks. The annotations consist of markers, scopes, loop bounds and generic constraints. Markers are used to provide unique names for each basic block, which can then be referred to by constraints. Constraints are inequalities that express relationships between the execution frequencies of basic blocks. Loop bounds are declared within a loop body and denote an upper bound for the execution count of the loop relative to the loop entry. Scopes are a mechanism to limit the area of validity of markers which allows us to express relationships that are local to a sub-graph of the ICFG.

3.2. Program Optimization and WCET Annotation Transformation

The FLOWTRANS phase deals with program sources which are already annotated by flow constraints. These can stem from either an earlier analysis pass or from a human. Flow constraints describe the control flow of the program in order to reduce the search space for feasible paths. These constraints, however, can be invalidated in the course of the compilation process by the application of optimizations that modify the control flow. This applies to optimizations such as loop unrolling, loop fusion and inlining, whereas optimizations such as constant folding and strength reduction do not affect the control flow. In order to ensure validity of the flow constraints throughout the compilation, a naive approach would be to disable control-flow modifying optimizations. This, however, would sacrifice the performance of the compiled code. As a part of TuBound, we thus implemented the FLOWTRANS component, a transformation framework for flow constraints which transforms the annotations according to the optimizations applied.

A large number of CFG-altering optimizations are loop transformations. For this reason, we based our implementation on the FORTRAN D loop optimizer that is part of ROSE. Keeping optimizations of interest separate from the compiler, our transformation framework is very flexible and also portable to other optimizers. The input of FLOWTRANS is an optimization trace (consisting of a list of all transformations the optimizer applied to the program) and a set of rules that describe the correct constraint update for each optimization. The concept of using an optimization trace can be applied to

Original annotated program

```
int* f(int* a)
{
        int i;
#pragma wcet_marker(m_func)
        for (i = 0; i < 48; i += 1) {
#pragma wcet_loopbound(48)
#pragma wcet_marker(m_for)
          if (test(a[i])) {
#pragma wcet_marker(m_if)
            // Domain-specific knowledge
#pragma wcet_restriction(m_if =< m_for/4)

            a[i]++;
          }
        }
        return a;
}
```

After loop unrolling by factor 2

```
int *f(int *a)
{
  int i;
  for (i = 0; i <= 47; i += 2) {
#pragma wcet_marker(m_f_1_1)
#pragma wcet_loopbound(24)
    if ((test(a[i]))) {
#pragma wcet_marker(m_f_1_1_1)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[i]++;
    }
    if ((test(a[1 + i]))) {
#pragma wcet_marker(m_f_1_1_2)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[1 + i]++;
    }
  }
  return a;
}
```

Figure 4. Prolog terms everywhere: WCET constraints before and after loop unrolling

any existing compiler. The rules need to be written only once per optimization. The rules, as well as the transformation of the flow constraints are written in Prolog and operate on the term representation of the AST. As a matter of fact, the syntax used to express the flow constraints is identical to that of Prolog terms, too, thus rendering the manipulation of flow constraints very easy. Figure 4 gives an example of such a transformation. We currently implemented rules for loop blocking, loop fusion and loop unrolling. With all support predicates, the definitions of the rules range from 2 (loop fusion) to 25 (loop unrolling) lines of Prolog [18].

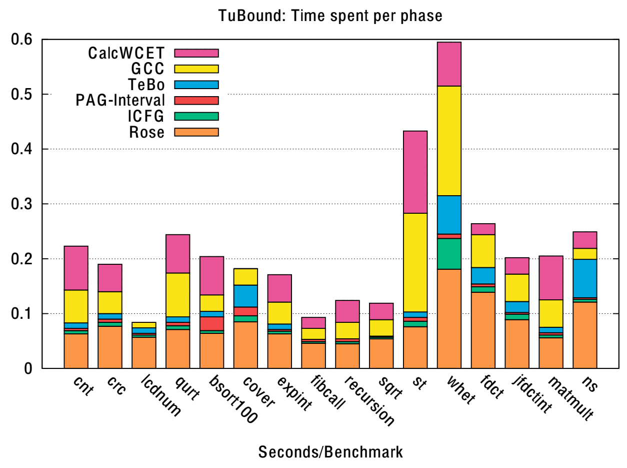
3.3. Compilation and WCET calculation

Compilation to Assembler Code. The annotated source code resulting from the previous stage is now converted into the slightly different syntax of the WCETC-language that is expected by the compiler [13]. This compiler is a customized version of GCC 2.7.2 which can parse WCETC and guarantees the preservation of all flow constraints at the C167 machine language level. The output of the GCC is annotated assembler code.

WCET Calculation. CALCWCET$_{167}$ reads the annotated assembler code that is produced by the GCC and generates the control flow graph of every function. CALCWCET$_{167}$ implements the IPET method and contains timing tables for the instruction set and memory of the supported hardware configurations which are used to construct a system of inequalities describing the weighted control flow graph of each function. The weights of the edges correspond to the execution time of each basic block. This system of inequalities is then used as input for an integer linear programming (ILP) solver that searches for the longest path through the weighted CFG. The resulting information can then be mapped back to the assembler code and can also be associated with the original source code.

4. Measurements

To demonstrate the practicality of our approach, we use a selection of benchmarks that were collected by the Real-Time Research Center at Mälardalen University [1]. For our experiments we selected those benchmarks that can be analysed by TUBOUND without annotating the sources manually. Figure 5 shows the time spent in the different phases of TUBOUND and the estimated WCET for a subset



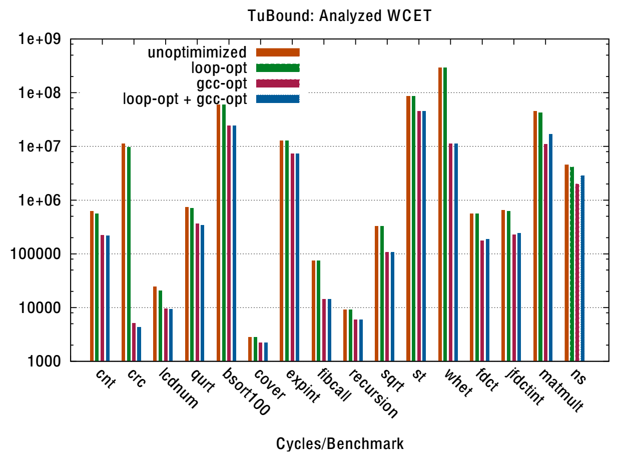


Figure 5. Analysis runtime (left) and analyzed WCET (right) of the selected benchmarks

of benchmarks. It must be noted that a large part (about 45% for the `ns` benchmark) of the time spent in TEBO is currently used to read and parse the term representation from one file and write it to another. This bottleneck can be eliminated by directly generating the data structure via the foreign function interface of the Prolog interpreter process and thus eliminating the expensive parsing and disk I/O. On the right-hand side of Figure 5 the influence of compiler optimizations on the WCET of the benchmarks can be seen, where the different bars per benchmark denote the analyzed WCET of the unoptimized program vs. the program with high-level and/or low-level optimizations turned on. Note that the y-axis uses a logarithmic scale. From the results, three different groups can be observed:

Group 1:	cnt, crc, lcdnum, qurt
Group 2:	bsort100, cover, expint, fibcall, recursion, sqrt, st, whet
Group 3:	fdct, jfdctint, matmult, ns

In the first group, the calculated WCET is always lower for the loop-optimized code. In the second group, the WCET is the same, regardless of loop optimizations. In the third group, the WCET of the loop-optimized program is better than that of the unoptimized program, however, if both kinds of optimizations are enabled, they interfere and less well performing code is generated, which is reflected by the higher WCET. One reason for this is extra spill code that is generated due to higher register pressure.

5. Conclusion

TUBOUND is a WCET analysis tool which is unique for combining the advantage of low level WCET analysis with high level source code annotations and optimizing compilation. The flow constraint transformation framework FLOWTRANS ensures that annotations are transformed according to the optimization trace as provided by the high-level optimizer. This approach allows us to close the gap between source code annotations and machine-specific WCET analysis. TUBOUND took also part in the WCET Tool Challenge 2008 [7], the results of which are published in [12].

Acknowledgements. We would like to thank Raimund Kirner for his support in integrating his tool CALCWCET$_{167}$ and Albrecht Kadlec for many related discussions.

References

[1] Benchmarks for WCET Analysis collected by Mälardalen University. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[2] The JTransformer framework. http://roots.iai.uni-bonn.de/research/jtransformer/.

[3] The program analyzer generator PAG. http://www.absint.com/pag/.

[4] The CALCWCET$_{167}$ tool. http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/.

[5] The ROSE Compiler. http://www.rosecompiler.org/.

[6] The static analysis tool integration engine SATIrE. http://www.complang.tuwien.ac.at/markus/satire/.

[7] The WCET tool challenge 2008. http://www.mrtc.mdh.se/projects/WCC08/.

[8] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems, Berlin, Germany*, June 1998.

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

[10] 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'06)*, Feb 2006.

[11] Christopher A. Healy, Mikael Sjodin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[12] Niklas Holsti, Jan Gustafsson, and Guillem Bernat (eds). WCET tool challenge 2008: Report. In *Proceedings 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, Prague, 2008.

[13] Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[14] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[15] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[16] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[17] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] Adrian Prantl. The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation. In Proceedings 14. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS'07)". Technical Report, Universität zu Lübeck, 2007.

[19] Peter Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[20] Markus Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. Proceedings 24th Workshop of „GI-Fachgruppe Programmiersprachen und Rechenkonzepte". Technical Report, Christian-Albrechts-Universität zu Kiel, 2007.

[21] Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In László Böszörményi and Peter Schojer, editors, *JMLC*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2003.

[22] Daniel Schulte. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Master's thesis, Universität Dortmund, 2007.

[23] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[24] Wankang Zhao, William C. Kreahling, David B. Whalley, Christopher A. Healy, and Frank Mueller. Improving WCET by Optimizing Worst-Case Paths. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 138–147, 2005.

WCET ANALYSIS FOR PREEMPTIVE SCHEDULING[1]

Sebastian Altmeyer[2], Gernot Gebhard[3]

Abstract
Hard real-time systems induce strict constraints on the timing of the task set. Validation of these timing constraints is thus a major challenge during the design of such a system. Whereas the derivation of timing guarantees must already be considered complex if tasks are running to completion, it gets even more complex if tasks are scheduled preemptively – especially due to caches, deployed to improve the average performance. In this paper we propose a new method to compute valid upper bounds on a task's worst case execution time (WCET). Our method approximates an optimal memory layout such that the set of possibly evicted cache-entries during preemption is minimized. This set then delivers information to bound the execution time of tasks under preemption in an adopted WCET analysis.

1. Introduction

Validation of hard real-time systems strongly relies on safe estimations of upper bounds on a task's worst case execution time (WCET). Computing such a WCET bound is already a complex problem for non-preemptively scheduled tasks. It becomes even more problematic in a preemptive environment. This means that the flexibility of a preemptive schedule comes at the cost of complex interaction between the tasks, such as preempting tasks evicting used data of preempted tasks out of the processor's cache. Nevertheless, some task-sets are only schedulable preemptively and, in addition to that, a non-preemptive schedule often exhibits a worse processor utilization. Thus, being able to compute both safe and precise WCET bounds for preemptive task-sets is essential.

For modern hardware architectures, however, Liu and Layland's assumption of negligible context switch costs [5] no longer holds. Instead, these costs often contribute substantially to the overall execution time, as Li et al. recently published in [4].

In this paper, we propose a new method which on the one hand decreases the context switch costs and on the other enables a precise and safe WCET analysis for preemptively scheduled tasks. Our method is an extension of the task mapping approach described in [1] which aims to increase the overall performance of a preemptive system. In our approach, we compute an arrangement of the tasks and its data in the memory such that the number of evicted cache entries of a task is minimized during preemption. The memory layout also induces a classification of the cache-entries which is then used to safely approximate WCETs under preemption. A major advantage of this approach is that both code and data remain unmodified, only the position in memory and thus in cache is changed.

The paper is structured as follows. In Section 2, we give a short intuition of our approach and the role

[1]This work was supported by the European Community's Sixth Framework Programme as part of ARTIST2 Network of Excellence and by the Seventh Framework Programme as part of PREDATOR.
[2]Universität des Saarlandes, Im Stadtwald 15, 66041 Saarbrücken, Germany
[3]AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

of different memory layouts. The optimization and analysis is described in Section 3, and compared to the related work in Section 4. Finally, Section 5 concludes this paper.

2. Memory Layout

The memory layout, i.e. the arrangement of data and instructions in the memory, determines the cache-sets to which data and instructions are mapped. Therefore, it strongly influences the cache interference and thus the context switch costs of preemptively scheduled periodic tasks.

Figure 1 depicts the correlation between the memory layout and the occupied cache-sets. A task-set with three tasks of size $n/2$ is scheduled such that only task 1 can preempt the other two. The system uses a direct mapped instruction cache of size n. In the first memory layout, if task 1 preempts task 3, the task might evict cache-entries of task 3 and thus induce context switch costs. In the second memory layout, no matter which task is preempted by task 1, no cache conflicts occur. A cache-set is called *endangered*, if it might be evicted during preemption and *persistent* otherwise. This notation, however, only relates to persistence during a single instance of a task, not to different instances of it. Hereby, tasks are seen as procedures periodically invoked by the scheduler. Note that finding a memory layout such that all cache-sets are persistent during preemption is impossible in general.

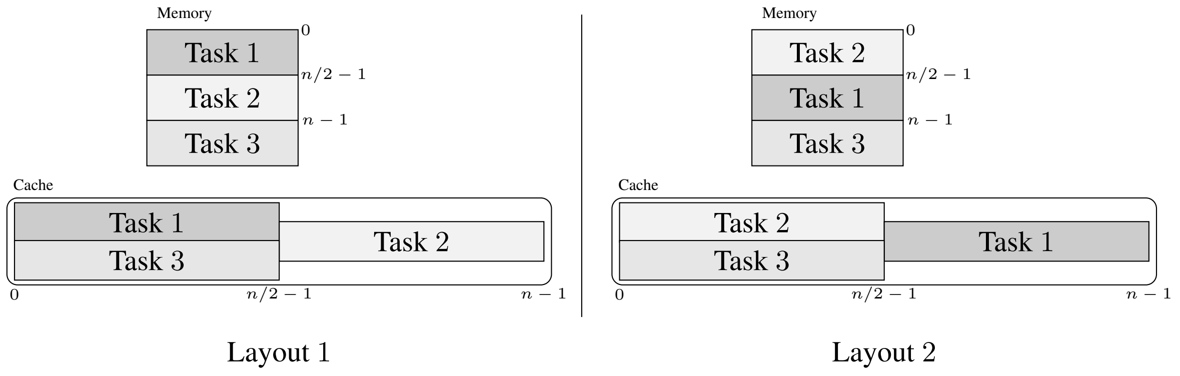


Figure 1. Two different memory layouts with different performance

3. Optimization and Analysis

In the following, we propose a combination of optimization and analysis of a memory layout in order to compute safe WCET bounds for preemptively scheduled tasks. First, we analyze the tasks to derive a metric to compare different memory layouts. We then approximate an optimal layout with respect to this metric and classify cache-entries as persistent or endangered. This classification is then used to compute safe WCET bounds for all tasks. The structure of the approach is shown in Figure 2.

In the remainder of this paper, we will use the following notation:

A cache is determined by the number of cache-sets m and the minimal life span k. The set of all cache-sets is denoted by $\mathbb{S}$. The minimal life span determines the minimum number of (read or write) accesses to a specific cache-set until the data of the first access may be evicted. This means that one can guarantee that after k different accesses, data of the first one is still cached, but after $k + 1$, one can not. A direct mapped cache has $k = 1$ since the second access (to the same set with different data) removes the data of the first. A 4-way LRU cache has $k = 4$ since a cache-set can hold data

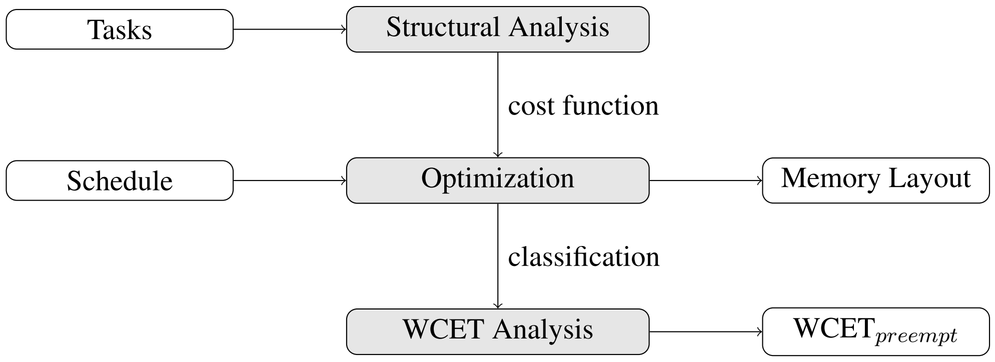


Figure 2. Overall structure of the approach

of 4 different accesses, but one further access will lead to eviction of data of the first one [7]. The analyzed hardware architecture may comprise either disjoint or unified instruction and/or data caches. Our approach is able to cope with both.

A task-set with n tasks is denoted with $T = \{\tau_1, \ldots, \tau_n\}$. The symbol τ_i denotes the task itself as well as the instructions of task τ_i. Each task has code size cs_i given in the number of cache-sets the code occupies[2]. In addition to the codesize, each task τ_i refers to a set of data fragment $D_i = \{d_{i,1}, \ldots, d_{i,l}\}$ where each such fragment has a size denoted with $ds_{i,j}$. The set of all data fragment of all tasks is denoted with $D = \bigcup_i D_i$. Data fragments refer to contiguous data blocks, arrays for instance, used by the tasks. The placement of these data blocks can be modified such that only the cache behavior (but not the semantics of program) is changed.

A task dependency relation $\vdash \subseteq T \times T$ determines the possible preemptions of the tasks. If $\tau_i \vdash \tau_j$ holds, task τ_i can preempt τ_j. Usually, the specification of communication channels or the assignment of static priorities implies such a dependency relation. For instance, with static priorities $Pr : T \rightarrow \mathbb{N}$ the relation is defined as: $\forall \tau_i, \tau_j, Pr(\tau_i) \leq Pr(\tau_j) : \tau_i \vdash \tau_j$. The relation $\vdash$ is reflexive to handle the fact that data may be evicted by other data of the same task.

A memory layout L maps code and data to start addresses in the memory. It is formally defined as

$$L : T \cup D \rightarrow \mathbb{S}$$

The start addresses are also given in the unit of cache-sets, i.e. modulo line-size. Although the function L allows empty fragments within the memory, i.e. parts which are not occupied by data or instructions, we only consider contiguous memory layouts.

The function

$$occ : (T \cup D) \times \mathbb{S} \rightarrow \mathbb{N}$$

determines how often a cache-set is occupied by a task or data fragment. For instance, if the cache has 128 sets and a task's code with size 129 starts at the first set, the first set is occupied twice (assuming usual modulo cache-mapping) and the others once. This function depends on a specific memory layout L, which we omit for the sake of simplicity of the notation.

The cost of a memory layout is determined by the possibly evicted cache-entry of all tasks. A cache-entry of a task τ_i may be evicted during preemption, if the same cache-set is occupied at least $k + 1$

[2]Note that we always refer to size as the size in number of cache-sets, i.e. $\lceil$size in bytes / size of a cache-line$\rceil$.

times by data of conflicting tasks ($\tau_j \vdash \tau_i$). Remember the definition of k. The cache can store data of k different accesses, one more access will lead to eviction.

The function $conf : (T \cup D) \times \mathbb{S} \mapsto \mathbb{N}$ defined as

$$conf(d_{i,j}, s) = \begin{cases} \sum\limits_{\tau_l \vdash \tau_i} occ(\tau_l, s) + \sum\limits_{\tau_l \vdash \tau, d \in D_l} occ(d, s) & \text{if } occ(d_{i,j}, s) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$conf(\tau_i, s) = \begin{cases} \sum\limits_{\tau_l \vdash \tau_i} occ(\tau_l, s) + \sum\limits_{\tau_l \vdash \tau, d \in D_l} occ(d, s) & \text{if } occ(\tau_i, s) > 0 \\ 0 & \text{otherwise} \end{cases}$$

returns the number of possible conflicts of task τ_i or data fragment $d_{i,j}$ in cache-set s.

The costs of a memory layout are thus computed by the sum over all tasks and all data fragments over all sets, where the number of conflicts exceeds k (which simply implies that these tasks or data fragments are considered endangered):

$$C = \sum_{x \in T \cup D} W(x) \left(\sum_{s=1}^{m} conf'(x, s) \right)$$

with

$$conf'(x, s) = \begin{cases} 1 & \text{if } conf(x, s) > k \\ 0 & \text{otherwise} \end{cases}$$

and a weighting function W which reflects a certain metric (as described in the following section).

3.1. Structural Analysis and Metric

The context switch costs are determined by the number of cache-sets which are 1) evicted during preemption and 2) reused by the preempted task. The memory layout shows only possibly evicted cache-sets. Which cache-set will be reused, and thus reloaded after preemption, depends on the structure of the task. If, for instance, each instruction of a task is executed at most once (during a single instance of that task), the context switch costs due to the instruction cache will be minimal. In contrast, loop structures may contribute significantly to the costs. Therefore, the pure number of evictions is not an appropriate metric to decide on an optimal memory layout. A simple metric that respects the task structure is to weight data depending on their maximal execution count.

Therefore, the structural analysis derives the following information needed for the cost function:

- size of tasks cs_i
- data fragments $d_{i,j}$ and size of data fragments $ds_{i,j}$
- weights

The size cs_i of a task τ_i can be read off the tasks directly. We employ a static analysis to derive the size of the accessed data, as follows: for a single access, the size is given as the width of the access;

for adjacent data, the accesses are combined to larger data fragments. Hereby, we assume that the targets of the memory accesses can be precisely computed (or at least overapproximated) statically.

The metric used to rate memory layouts is strongly determined by the weight function W, which is defined as follows:

$$W : T \cup D \to \mathbb{N}$$

The weight function has to be chosen such that the eviction of data with low weight has minor impact to the context switch costs than the eviction of data with high weight.

To accomplish this, we weight each data fragments in the following way:

$$W(d_{i,j}) = \begin{cases} 2n & \text{if } d_{i,j} \text{ is accessed in a loop} \\ 1 & \text{otherwise} \end{cases}$$

$$W(\tau_i) = n$$

Depending on the program structure, i.e. if a data fragment is accessed in a loop or not, the weights are assigned to the data fragments. The weight function assigns the value n to all instructions of all tasks. The number $2n$ for the data fragments is chosen to ensure that the eviction of a loop fragment weighs more than the eviction of non-recurring code or memory accesses and thus becomes much more unlikely (as previously defined, n is number of tasks). Note that the current weight function is only preliminary and still has to be evaluated. Further analyses of the structure of the tasks can be used to increase the accuracy of the metric.

3.2. Optimization

The next step is to find an optimal memory layout, or, at least, a near-optimal layout. Remember that we restrict the method to contiguous memory layouts, i.e. memory layouts without empty spaces. This means that such a memory layout is described by a sequence of tasks and data fragments: element x_i starts at the end of the preceding element x_{i-1}, i.e.

$$\forall x \in (T \cup D) : L(x_i) = L(x_{i-1}) + \begin{cases} cs_{i-1}, & \text{if } x_{i-1} \in T \\ ds_{i-1}, & \text{if } x_{i-1} \in D \end{cases}$$

Due to this restriction, all memory layouts are permutations of an initial layout. Such a permutation is denoted with the symbol σ and C_σ denotes the costs of the memory layout described by the permutation σ. A permutation σ' is a neighbor of permutation σ, iff σ' can be reached from σ by swapping the position of two elements within σ. The set of all neighbors of permutation σ is denoted by $Ne(\sigma)$.

To approximate an optimal memory layout we are using hill-climbing as shown in Figure 3: the algorithm starts with a random permutation σ_{start}. It then selects the neighbor of σ_{start} with the lowest costs and continues searching an optimal layout from this element on. In case no further improvement is possible, the algorithm may select the second best result to explore a larger portion of the state space. A predefined parameter p restricts the number of times the algorithm selects a second best permutation. By this, the parameter P can be adjusted to treat precision against running time of the algorithm. The set $visited$ keeps track of all already seen permutations to ensure that each element is visited at most once.

```
hill_climbing (permutation $\sigma_{start}$, unsigned int $p$)
{
  $\sigma_{cur} = \sigma_{start}$
  $\sigma_{best} = \sigma_{start}$
  visited = $\{\sigma_{start}\}$
  **while** ($p > 0$) {
    $/*$ select next candidate $*/$
    **let** $\sigma' \in Ne(\sigma_{cur})$
    **with** $C_{\sigma'} = \min(\{C_\sigma | \sigma \in Ne(\sigma_{cur}) \setminus \text{visited}\})$
    visited = visited $\cup \{\sigma'\}$
    $/*$ is it better than the current? $*/$
    **if** ($C_{\sigma_{cur}} > C_{\sigma'}$) {
      $/*$ is it best permutation seen so far? $*/$
      $\sigma_{cur} = \sigma'$
      **if** ($C_{\sigma_{best}} > C_{\sigma'}$) { $\sigma_{best} = \sigma'$ }
    }
    $/*$ if not, continue with a worse result $*/$
    **else** {
      $p = p - 1$
      $\sigma_{cur} = \sigma'$
    }
} }
```

Figure 3. Hill climbing to compute an optimal memory layout

3.3. WCET Analysis using Cache Classification

The memory layout induces a classification on all memory accesses of all tasks: a cache-entry of a task τ_i is either persistent (in case the number of conflicts is less than or equal to k) or it is endangered.

$$\text{classify(x, s)} = \begin{cases} \text{endangered} & \text{if } conf(x, s) > k \\ \text{persistent} & \text{otherwise} \end{cases}$$

This classification can be easily used to compute a safe WCET bound under preemption. If a low-level analysis detects an access to an endangered cache-set, the analysis has to handle both cases: cache-miss and cache-hit. In case of an access to a persistent cache-entry, the analysis behaves as usual. The computed WCET bound is valid, even if the task is preempted; only the endangered cache-sets might be invalidated. Cache-related timing anomalies are also treated correctly: the analysis assumes both cases and thus computes a valid WCET bound even if a cache-hit might result in a higher execution time than a cache-miss.

4. Related Work

Cache partitioning to prevent task interference during preemption has been proposed by Mueller [6] and Wolfe [14]. The cache is divided into uniform segments such that each task operates on it own. Hereby, tasks can not interfere on the cache and thus, the WCET analysis for non-preemptive systems can be used. In contrast to our method, not only the memory layout and thus the memory-to-cache mapping is adapted, but also the code itself underlies major modifications; in order to adapt to code and data to fit into its cache segment, new branches and computation for data accesses have to be introduced. This, in addition to the highly decreased cache-size for each task, impairs the performance of the system even more.

A WCET analysis for preemptive system has been proposed by Schneider [9]. In his approach, a preemption is possible and thus assumed at every program point of the analyzed task. Thus, the analysis considers each cache access to be unknown. Obviously, the analysis computes safe approximations of the WCETs of preemptively scheduled tasks. The precision of the approach, however, suffers from this highly pessimistic assumption. Compared to our approach, we can classify cache-sets as persistent and can thus reduce this overapproximation.

The most eminent approach to timing analysis for preemptive systems computes the context switch costs separately from the WCET bound of a task. Hereby, the notion of useful cache blocks (UCBs) introduced by Lee et al. [3] plays a major role. A cache block is *useful* at a certain program point, if it may be accessed again after this point. Thus, if the cache block is evicted during preemption, the program may need to reload it and the time for this contributes to the context switch costs. Staschulat et al. [11] and Tan et al. [13] extended this approach to increase the precision (mainly by computing the set of possibly evicted cache-set in addition to the UCBs). Adapted schedulability analyses that incorporate context switch costs have been proposed in [12] for static priorities and in [2] for dynamic priorities. The main difference to our approach is on the one hand the optimization of the memory layout – which also could reduce the number of useful cache blocks – and on the other hand the handling of cache-related timing anomalies [8]. Only counting the time needed to reload cache-sets does not obey the fact that a cache-hit may lead to a higher execution time than a cache-miss. In our approach, the WCET analysis directly incorporates the cache-set classification and thus derives safe upper bound (also in the presence of timing anomalies). In addition, the usual schedulability analyses can be applied.

Other approaches, by Sebek [10] for instance, rely on measurement. However, even for a single task, full coverage is hardly achievable. Preemptive scheduling introduces an even higher level of complexity, rendering measurement-based approaches nearly infeasible – at least, no guarantee that the measured execution times deliver safe upper bounds can be given.

5. Conclusion and Future Work

In this paper, we propose a new approach to optimize and analyze the WCET of preemptively scheduled tasks. Our approach uses the fact that different memory layouts can lead to vastly different context switch costs. We first derive a metric to rate memory layouts and then approximate an optimal one. Such a memory layout induces a classification of the cache-entries into *endangered* or *persistent*. This information is then incorporated in a traditional WCET analysis allowing the analysis to derive both safe and precise worst case execution time bounds of preemptively scheduled tasks.

In the future, we plan to implement the whole toolchain and to provide an evaluation of the presented approach. To further improve the precision of the approach, a more fine-grained metric could provide more accurate cache-entry classification, e.g., by taking a maximal execution count of instructions into account. The current approach only copes with preemption occurring at arbitrary program points. Thus, an analysis of the whole schedule could provide more details about the task-set (preemption points), allowing our approach to compute a tighter set of endangered cache-entries.

References

[1] Gernot Gebhard and Sebastian Altmeyer. Optimal Task Placement to Improve Cache Performance. In *Proceedings of the 7th ACM Conference on Embedded Systems Software (EMSOFT'07)*, pages 259–268, October 2007.

[2] Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1623–1628, San Jose, USA, 2007.

[3] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyne Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong San Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[4] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[6] Frank Mueller. Compiler support for software-based cache partitioning. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133, 1995.

[7] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Predictability of Cache Replacement Policies. Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS, September 2006.

[8] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[9] Jörn Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium 2000*, pages 195–204, November 2000.

[10] Filip Sebek. Measuring cache related pre-emption delay on a multiprocessor real-time system. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, 2001.

[11] Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for preemptive scheduling. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 157–165, New York, NY, USA, 2005. ACM.

[12] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 41–48, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1):7, 2007.

[14] Andrew Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computing and Software Engineering*, 2(3):315–327, 1994.