9th International Workshop on Worst-Case Execution Time Analysis

WCET 2009, June 30, 2009, Trinity College, Dublin, Ireland

Edited by

Niklas Holsti

Editor
Niklas Holsti
Tidorum Ltd
Tiirasaarentie 32
00200 Helsinki, Finland
niklas.holsti@tidorum.fi

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

ISBN 978-3-939897-14-9

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

Publication date
November, 2009.

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.

License

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works license: http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the author's moral rights:

- Attribution: The work must be attributed to its authors.
- Noncommercial: The work may not be used for commercial purposes.
- No derivation: It is not allowed to alter or transform this work.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.WCET.2009.i

ISBN 978-3-939897-14-9 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

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

ISSN 2190-6807

www.dagstuhl.de/oasics

Preface to WCET'09 Proceedings

On June 30, 2009, thirty-five people from nine countries and three continents met in Trinity College, Dublin, to hold the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09, http://www.artist-embedded.org/artist/WCET-2009.html). The workshop was organised as a satellite event of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09, http://ecrts09.dsg.cs.tcd.ie).

The goal of this annual workshop is to bring together people from academia, tool vendors, and tool users in industry who are interested in all aspects of timing analysis for real-time systems. The workshop features a highly interactive format with ample time for in-depth discussions. Topics of interest include:

- Different approaches to WCET computation
- Flow analysis for WCET, loop bounds, feasible paths
- Low-level timing analysis, modeling and analysis of processor features
- Strategies to reduce the complexity of WCET analysis
- Integration of WCET and schedulability analysis
- Evaluation, case studies, benchmarks
- Measurement-based WCET analysis
- Tools for WCET analysis
- Program and processor design for timing predictability
- Integration of WCET analysis in development processes
- Compiler optimizations for worst-case paths
- WCET analysis for multi-threaded and multi-core systems.

The papers presented at the workshop were selected based on peer reviews by program comnmittee members and external reviewers, all experts in the field. The final proceedings of the workshop contain the presented papers, updated in response to the discussion at the workshop, the abstract of the invited talk by prof. Petru Eles, and a summary of the panel discussion that concluded the workshop. These final proceedings thus update the pre-proceedings that were distributed to the workshop participants for the workshop.

I am happy to thank the authors, the Program Committee including the external reviewers, the WCET Workshop Steering Committe, and the ECRTS'09 organizers for assembling the components of a very stimulating workshop. The workshop organizers are also deeply grateful to the ArtistDesign Network of Excellence (http://www.artist-embedded.org/artist/) for financial support and to the Dagstuhl DROPS archive for making the final proceedings available on-line. The slide presentations are not available on DROPS, but can be retrieved from the ArtistDesign site, http://www.artist-embedded.org/artist/WCET-2009.html, as a single ZIP archive, or separately for each paper from the "Program" tab.

Niklas Holsti
chair, WCET'09
Tidorum Ltd

10 November 2009

TEACHING WCET ANALYSIS IN ACADEMIA AND INDUSTRY: PANEL DISCUSSION

Niklas Holsti (ed.)[1]

1. Introduction

The last item on the programme of the WCET'09 workshop was a panel discussion on "Teaching WCET analysis in academia and industry". The panelists presented three position statements to initiate a general discussion of the subject. This summary is based on the text of the position statements that the panelists gave me, in my role as the panel chair, on the panelists' presentations, and on my notes of the discussion.

The four experts who kindly agreed to form the panel (of whom Reinhard Wilhelm regrettably could not attend the workshop) span the field from academic research in program analysis to commercial developers and providers of WCET tools. Both static analysis methods and measurement-based methods are represented. However, the end users of WCET analysis tools are not directly represented.

As panel chair I asked the panelists to address some or all of the following questions in their position statements:

- Place, time, and role of WCET analysis in the academic curriculum for real-time systems.
- Contrasts between teaching WCET analysis in general to students, and teaching or training professionals to use a specific WCET tool.
- Among the current professional users of WCET analysis, was anyone taught WCET analysis while a student?
- Common misconceptions and mistakes about WCET analysis and WCET tools, among students and professionals, and how to correct them.
- Which issues in WCET analysis are important? Differences between the academic and industrial assumptions and points of view.
- Stupid questions that students and users ask – and are they really so stupid, or are they deep?
- Teaching materials: texts, presentations, exercises, bibliographies. Can they be shared?
- Making WCET analysis and WCET tools more teachable and understandable: ideas and directions.

Section 2 of this summary presents the panelists' position statements and section 3 summarises the ensuing discussion.

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

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2278

2. Panelist Position Statements

2.1 Peter Puschner (Vienna University of Technology)

The Vienna University of Technology teaches WCET analysis in an optional course that consists of 15 hours of lectures and at least 10 hours of laboratory exercises. A course in Real-Time Systems is a prerequisite; that course is held by Hermann Kopetz and focuses mainly on the Time-Triggered Architecture (TTA). The goals of the WCET-analysis course are to show that WCET analysis is not trivial; to correct misconceptions of WCET analysis; and to show how to achieve predictability of real-time software. The laboratory exercises use a processor simulator and a scheduling simulator. The course covers pure WCET analysis, the relationship of WCET and scheduling (real-time versus non-real-time), and writing time-predictable, WCET-oriented code. The course is taken by about eight students each year, in their fourth or later year of study.

We observe that students are fascinated by this "unknown", multi-faceted field that involves program analysis, programming, programming languages, compilers, computer architectures, and so on. However, "thinking predictability" seems to be very difficult for students, and working with processor simulators is unsatisfactory, except for experimenting with scheduling. On the positive side, our experience shows that WCET-oriented programming can really improve worst-case performance.

The WCET problem is still widely unknown. Moreover, the complexity of WCET analysis and timing analysis in general is under-estimated. For example, why do so few people care about timing interferences, i.e., influences on task timing that are due to the fact that the execution of each task changes the state of shared resources and thus the timing of other tasks? In our WCET course the trainees learn about complex influences on the timing, about (global) timing interferences, and about predictability, both about obstacles to predictability and how to achieve predictability. In conclusion, I suggest that WCET training should be mandatory for all students in Embedded Systems.

2.2 Christian Ferdinand (AbsInt GmbH) and Reinhard Wilhelm (Saarland University)

For AbsInt as a company it is important that students understand that determining safe upper bounds of the worst-case execution time is an interesting and relevant problem in programming real-time applications and that there are solutions for this problem in form of tools (preferably from AbsInt).

WCET research in Saarbrücken started at the Compiler and Programming Languages chair. The underlying principles of program analysis are taught in “Compiler Construction” and “Program Analysis” courses. The specific analysis problems of WCET analysis and especially cache behavior prediction is covered in 2-4 hours. The core course “Embedded Systems” devotes 2 hours to WCET analysis. A subsequent course “Embedded System Design” has a more practical focus. Students learn how (safety-critical) embedded real-time applications are developed with the help of modern development tools. In the practical part the students develop the control software for a *Lego Mindstorm* robot toy car that follows automatically a lane. Used are the *SCADE* Suite of Esterel

Technologies, the real-time operating system *nxtOSEK*, *aiT* WCET analysis tool, and the scheduling analysis tool *Symta/S* of Symtavision.

WCET analysis has been and is being taught at various advanced courses and Summer Schools, ARTIST DESIGN school, Onassis School on Crete. aiT tool usage is covered in a "standard" 2 days course for aiT users. AbsInt also regularly offers presentations and tutorials about aiT at various conferences/trade shows. We higly welcome efforts by universities to teach WCET analysis. Since the focus (underlying theory/tool usage) and available teaching times vary widely, we usually put together a set of slides for each request and provide aiT license keys for our standard targets. Dear teachers, please do not hesitate to contact us (*info@absint.com*).

As reading material, we recommend the chapter on WCET analysis in the *Embedded Systems Handbook*, Taylor & Francis (Ed. R. Zurawski) and/or R. Wilhelm *et al*. "The worst-case execution-time problem-overview of methods and survey of tools" (*Trans. on Embedded Computing Sys*., 7(3):1–53, 2008). Recorded lectures of Reinhard Wilhelm are also available at *http://www.artist-embedded.org/docs/Events/2008/Autrans/Videos/Reinhard_Wilhelm/* and *http://www.forth.gr/onassis/lectures/2008-07-21/programme.htm.*

2.3 Guillem Bernat (Rapita Systems Ltd and University of York)

The University of York teaches WCET analysis in an elective module on Real-Time Systems in the last year of degree study. WCET analysis is covered in one or two lecture hours within the total of 18 hours for this module. There was no laboratory work in 2009. The goal is to make the students understand the issues that affect execution-time behaviour; only then can one address the problem of WCET calculation. We strive to impart a general understanding of how to build analysable systems and show how to move away from an "average case execution time" way of thinking towards WCET-analysable designs.

As preliminaries, we assume knowledge of assembly language, processor architecture, scheduling theory (explained during the Real-Time Systems module), and programming languages and program architectures for real-time software. The course does not address any specific target processor. It gives an overview of WCET-analysis techniques and tools, with focus on the RapiTime tool architecture. Some motivational examples are given but detailed examples are left for self-study. So far, we know of no former student of this course who has then used WCET analysis professionally. However, students tell us that the course is useful and has made them realise the complexity of the issues involved in timing analysis.

3. Discussion

The discussion was not recorded, nor were speakers asked to identify themselves, so this summary does not try to attribute comments to specific speakers.

As could be expected for this venue, the need to teach WCET analysis as part of the curriculum for real-time and embedded systems was not questioned. Instead, the discussion had two main foci: the problem of the "average case" mind-set, and finding good examples and laboratory exercises for WCET analysis.

It was generally agreed that WCET analysis is not for early students. Understanding WCET analysis needs background knowledge of processor architectures, assembly languages, real-time systems and scheduling, and perhaps some understanding of program analysis. But the early mainline courses in programming tend to teach students to optimize programs for the average case. The later courses in real-time programming and WCET analysis must then work to erase this average-case mind-set.

It was pointed out that there is one mainline computing domain that has real-time constraints: computer graphics and games. The image rendering algorithms and data structures must be designed to bound the worst-case performance, else the frame-rate can drop suddenly and very observably in some cases, for example when the viewpoint changes. Perhaps some ideas of WCET analysis could be introduced in computer graphics courses.

For examples and laboratory exercises in WCET analysis, work with real devices such as the Lego robotics kit is very motivating, but their fascination can also be distracting. Furthermore, the control systems in real examples are often so robust that a deadline miss has no bad effects, leading at most to a small hiccup in the visible behaviour. This is teaching the wrong lesson! A deadline miss should have a dramatic effect. As an example, a computer-controlled mouse-trap was suggested: the student inserts a finger to trigger the trap; the software detects that the catch is released and must stop the steel before it strikes the finger...

A GENERIC FRAMEWORK FOR BLACKBOX COMPONENTS IN WCET COMPUTATION

C. Ballabriga, H. Cassé, M. De Michiel[1]

Abstract

Validation of embedded hard real-time systems requires the computation of the Worst Case Execution Time (WCET). Although these systems make more and more use of Components Off The Shelf (COTS), the current WCET computation methods are usually applied to whole programs: these analysis methods require access to the whole system code, that is incompatible with the use of COTS. In this paper, after discussing the specific cases of the loop bounds estimation and the instruction cache analysis, we show in a generic way how static analysis involved in WCET computation can be pre-computed on COTS in order to obtain component partial results. These partial results can be distributed with the COTS, in order to compute the WCET in the context of a full application. We describe also the information items to include in the partial result, and we propose an XML exchange format to represent these data. Additionally, we show that the partial analysis enables us to reduce the analysis time while introducing very little pessimism.

1. Introduction

Critical hard real-time systems are composed of tasks which must imperatively finish before their deadline. To guarantee this, a task scheduling analysis, that requires the knowledge of each task WCET, is performed. To compute the WCET, we must take into account (1) the control flow of the task (flow analysis) and (2) the host hardware (timing analysis). The current WCET computation methods are designed to be used on a whole program. However, there is some drawbacks to this approach. First, the analyses used for WCET computation usually run in exponential time with respect to the program size. Second, when the program to analyze depends on external components (e.g. COTS or libraries) whose sources are not available, the lack of information about the components prevents the WCET computation if information from the user is requested.

This paper focuses on the last problem, but it is shown that, as a side-effect, the computation time is also decreased. This article presents a generic method to partially analyze a black-box component, producing a component partial result that may be instantiated at each component call point to produce actual analysis results. We also define the information items that must be included in the component partial result, and propose an XML exchange file format.

As WCET computation involves a lot of analyses, the first section presents two of them as examples, the instruction cache analysis and the loop bound estimation, and show how to adapt them to be able to do partial analysis. Then, in the next section, we generalize our approach based on the previously described examples. We present our implementation (XML exchange format definition, and experimental results with OTAWA) in the third section and, after presenting related works in section 5,

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

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2290

we conclude in the last section.

2. Component analysis

To compute the WCET of a program, one must previously perform a lot of analyses, some related to the flow control analysis (infeasible paths detection, loop bounds evaluation, etc), other related to the architecture effects analysis (cache analysis, branch predictor analysis, etc). The actual WCET computation can be done in several ways, but a widely used approach is IPET [14] (*Implicit Path Enumeration Technique).* This technique assigns a variable to each *Basic Block* (BB) and edge, representing the number of executions on the path producing the WCET and builds a linear constraint system with these variables to represent the program structure and the flow facts (such as loop bounds). The WCET is then the maximization of an objective function depending on these variables and on the individual execution time of each BB computed by an ILP (Integer Linear Programming) solver.

To explain the general principles of component analysis, we study a simple scenario where a program `main()` is calling a function `callee()` in a component. Well known analyses involved in WCET are used: the instruction cache behavior analysis and the loop bounds analysis. For each one, we show how to partially analyse the `callee()` function to produce a partial result PR_{callee}. Then we show how to analyse the program `main()` while composing informations from PR_{callee} (but without any access to the `callee()` code), to obtain analysis results for the whole program.

2.1. Instruction Cache Analysis

The partial cache analysis method presented in this section is based on the whole-program instruction cache[1] analysis presented by [12, 1] and improved in [2], which computes, by static analysis, a category describing a particular cache behavior for each instruction. The instructions whose execution always result in a cache hit (resp. miss) are categorized as *Always Hit* (AH) (resp. *Always Miss* - AM). A third category, *Persistent* (PS), exists for the instructions whose first reference may result in a cache miss, while the subsequent references result in cache hits. Other cases are set to *Not Classified* (NC). From these categories, new ILP constraints are added to the ILP system.

The categories are computed by abstract interpretation [8, 9] on a *Control Flow Graph* (CFG), composed of BB. *Abstract Cache States* (ACS) are computed before and after each BB which are in turn used to compute the categories. Three different analyses are done, each one using its own kind of ACS: the *May*, the *Must*, and *Persistence* analyses. While the *May* (resp. *Must*) analyses computes the cache blocks that may (resp. must) be in the cache, the *Persistence* finds the persistent blocks.

As previously stated in [4], three issues exist when dealing with partial instruction cache analysis. First, when executed, the `callee()` function ages and/or evicts cache blocks from the main program and so, it affects the categorization of the main program blocks. This is modeled by a cache *transfer* function (based on previous works by A. Rakib et al. in [18]) for each cache analysis. This *transfer* function is defined by $ACS_{type}^{after} = transfer(ACS_{type}^{before})$ (where *type* is one of *may*, *must*, or *pers).* In other words, it gives, for each cache analysis, the ACS after the function call based on the ACS before. Second, the `callee()` function contains blocks that need to be categorized. Unfortunately, the categorization of these blocks is not constant, but depends on the ACS at the function calling site.

[1] In this paper, we suppose that we have a set-associative instruction cache with *Least Recently Used* (LRU) replacement policy.

So we have introduced in [4] a *summary* function (only one is necessary to handle the three analyses) that is defined by $cat = summary(ACS_{must}^{before}, ACS_{may}^{before}, ACS_{pers}^{before})$, where *cat* is a Conditional Category (CC), i.e. a function such that $cat : Block \rightarrow \{ AH \,|\, AM \,|\, PS \,|\, NC \}$. In other words, it takes the various ACS before the call, and returns a category for each BB of `callee()`.

The last problem is that, when doing the partial analysis on `callee()`, the component base address is not known while it dictates which cache lines are affected by ageing, and which blocks are loaded. To take this into account, we force the alignment of cache blocks boundaries on `callee()` to be the same at partial analysis time and at instantiation time by enforcing the actual base address to be aligned on cache block size. Then, the matching between line results at partial analysis time and instantiation time is just a matter of shift and roll on the whole set of lines. This constraint may be easily achieved with most linkers.

Once we have the `callee()` partial result, the composition with the main program is done in two steps (illustrated in figure 1), without any further access to the `callee()` source code or executable:

1. We perform the instruction cache analysis on the main program in the same way as we would have done on a traditional (monolithic) analysis, except that we model the calls to the `callee()` function by the *transfer* function. At the end of this step, we obtain the ACS before and after each BB of `main()`.
2. Next, as we know the ACS values for `main()`, we have the ACS at the calling site of `callee()` and, therefore we can apply the *summary* function to obtain the categorization of `callee()` blocks.

At the end, we have got the categorization for all BB of the program (including component BB).

2.2. Loop Bound Analysis

The loop bound partial analysis presented in this paper is based on works of [10]. It tries to estimates the *max* and *total* iteration count for each loop in the program (the *max* is the maximum number of iterations for one entry into the loop, while the *total* is the total number of iterations of the loop during the execution of the outermost loop). The loop bound estimation is performed in three steps:

1. First, a context tree representing the program is built (a context tree is a tree where nodes are loops or function calls, and where children nodes are inner loops or function calls), and each loop is represented by a *normal form.* This normal form enables us (if possible) to get an expression representing the maximum iteration number of the loop, parametrized by the context of the loop. The computation uses abstract interpretation, in which the domain is an *Abstract Store (AS)* that maps each variable in the program to a symbolic expression describing its value.
2. The second step performs a bottom-up traversal of the context tree and instantiates each parametrized expression of *max* and *total* (computed in first step) to get absolute, context-dependant expressions for each loop.
3. Although the second step computes *max* and *total* expressions for each loops, it remains complex expressions that are evaluated in this last step to get a numerical value. Finally, a tree annotated with the results for each loop is returned by the analysis.

To perform partial loop bound analysis, we need (1) to determine the effect of the call to `callee()` on the main program (side effects, modified global variables, reference-passed arguments), and (2) to supply parametrized loop bound expressions depending on the calling context.

In fact, the analysis proposed in [10] is well suited to perform partial analysis. For the sub-problem (1), the *Abstract Stores* represents values of the program variables as symbolic expressions, that is, they support free variables representing data provided in the caller context. Therefore, performing the transfer function is just replacing the context variables by their actual values in the *Abstract Store* at the exit of the component. This allows us to satisfy the sub-problem (1), but it can be refined: to avoid wasting memory and computation time, we can trim the component *Abstract Store* by eliminating the variables that cannot have an effect on the main program. Only the remainder is part of the partial result of `callee()`.

To handle the sub-problem (2), we have to analyze the `callee()` function on its own (as if it was the main entry point). We need to perform the analysis up to (and including) step (2) that provides the symbolic expressions of the *max* and *total* for all loops in `callee()`. Then, it is easy to apply a context *Abstract Store* to this symbolic values at composition time by replacing free variables by their actual values. These symbolic expressions are stored in the partial result computed for `callee()`.

To sum up: the partial result for `callee()` contains (1) the (trimmed) component *Abstract Store* playing role of the *transfer* function and (2) the symbolic expressions of the *max* and *total*, used as a *summary* of the component. Likewise to the cache partial analysis, the obtained partial results are composed in two steps:

1. First, loop bound analysis is performed on the main program, while modeling `callee()` by the *Abstract Store*. This enables us to know the *max* and *total* bounds for the loops in `main()`.
2. Second, as we have got the calling context of `callee()`, we can apply it to the symbolic expressions of its loop bounds.

We can see a lot of similarities between the partial cache analysis and the partial loop bound analysis approaches. The next section proposes a generalization to other analyses.

3. Generic Principles of Component Analysis

A component is a block of code with one or more component entry points. When the analysis is performed, there is a partial analysis for each entry point, each one giving a partial result, and their aggregation is the partial result of the entry point. While in the last section, we have surveyed some specific examples of partial analyses, we give here a more general approach of component analysis.

3.1. Distribution Model

Let's consider an example where a black-box component (a COTS) is developed by a so-called *Producer*. The black-box component is delivered to a so-called *Assembler* that develops the main program using, among other things, the COTS and assembles them into the final executable. The main program contains hard real-time tasks, and so their WCET must be known. Even if the *Assembler* has no access to the COTS code, components must be analyzed and the *Producer* must compute the

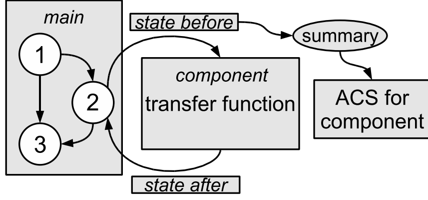


Figure 1. Partial cache analysis composition.

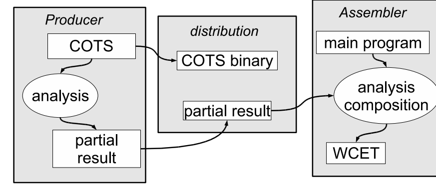
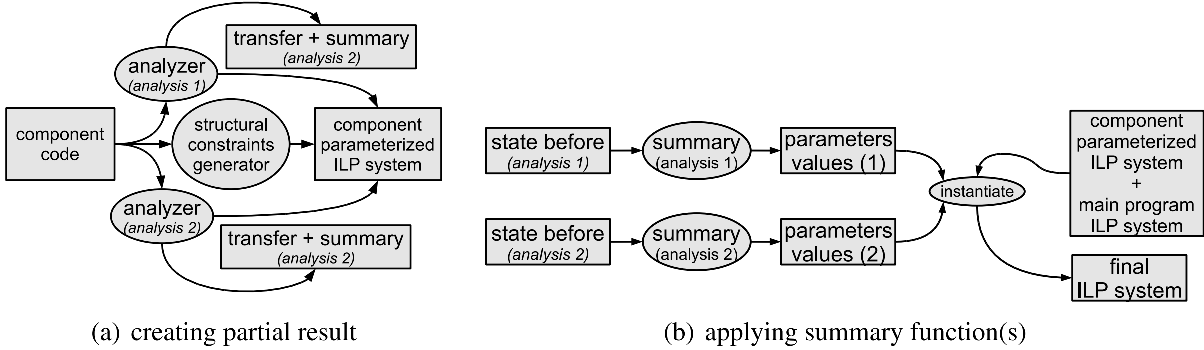


Figure 2. Distribution model.



(a) creating partial result

(b) applying summary function(s)

Figure 3. partial result creation and use

component partial result of its COTS, and delivers it with the COTS binaries, to let the *Assembler* compute the WCET of its application (figure 2).

In the examples of the previous section, we have seen that there is three sub-problems: (1) the fact that the called COTS function has an effect on the main program, (2) the call site dependencies of the COTS function analysis results, and (3) the variability of the COTS base address.

3.2. Transfer and Summary Functions

A lot of analyses involved in WCET computation produces abstract *states* at specific program points and then derives properties useful to the WCET computation. For example, we have several cache analysis methods (see [12] and [15] for two different techniques), loop bound analysis [10], branch prediction analysis [7], etc.

For this kind of analyses, the sub-problem (1) can always be handled by a *transfer* function, associated with each component entry point function. This *transfer* function is a generalization of the cache *transfer* function and is defined such that $state^{after} = transfer(state^{before})$, giving state after the call as a function of the state before the call. The terms $state^{after}$ and $state^{before}$ are of the same type, that is, dependent on the analysis and corresponding to the analysis domain. If the domain of the analysis is not the same for the COTS and for the main program, then the *transfer* function must be extensible (adaptable) so that, when we try to use it at composition time, its signature matches the domain used for the main program (the same is true for the generalized *summary* function).

The sub-problem (2) can be handled by a *summary* function that is a generalization of the *summary* function presented in previous section, with some extensions to model the contribution to the ILP system used for IPET.

While the generalization of the *transfer* function was quite straightforward, the *summary* function needs to consider the fact that a static analysis does essentially two things: (1) it computes states at some program points, and (2) it interprets the resulting states to deduce useful information for WCET computation. The state type, the deduction method and the resulting WCET-related informations are analysis-specific, but all analysis results contribute to the ILP system: some generalization may be possible for the *summary* function.

To this end, we define a parametrized system, and describe the results of the *summary* function as its instantiation in function of actual parameter values. The parameters are integer values with symbolic names. In the parametrized ILP system, each term in a constraint can optionally contain one parameter factor per term (ex: $2x_1.par_1 + 3.x_2 = 5.par_2$). We may also refine our parametrized ILP system by adding conditional rules with condition depending on the parameters that may activates some constraints. For example if a category of block 1 is always hit, we add a constraint specifying that number of miss is 0: $if(par_1 = AH)\ \{x_1^{mis} = 0\}$.

For each component entry point and for each analysis, the partial result includes a set of parameters describing the WCET-related properties depending on the calling context. In turn, the parameters are applied to a parametrized ILP sub-system describing the contribution of the component to the WCET. Each possible calling context can be represented by a valuation of the parameters. The *summary* function takes $state^{before}$ as parameters, and return a list of $(name, value)$ pairs for each parameter: $< (n_1, p_1), (n_2, p_2), ..., (n_k, p_k) >= summary(state^{before})$, where the p_i are the parameter values and n_i are the parameter names. The parametrized ILP sub-system contains common parts like the structural constraints, and analysis-specific parts (i.e. the contribution of each specific analyzer).

To sum it up, the component partial result (whose creation is outlined in figure 3 (a)) contains, for each component entry point function, (1) for each analysis, the *transfer* function taking the analysis state before the call, and giving the state after, (2) for each analysis, the *summary* function, which takes the analysis state before the call to the function, and gives back the values of the parameters, and (3) the partial parametrized ILP system describing the effect of the entry point on the WCET.

To instantiate this partial result (i.e. to find the WCET contribution associated with this component entry point function given the context), we need (1) to analyse the main program while modeling the component function by the *transfer* function, (2) apply the *summary* function to find the values of the parameters, that allows us (3) to instantiate the parametrized ILP sub-system representing the component. This ILP sub-system is then merged with the ILP system of the main program, allowing us to compute the WCET (see figure 3 (b)).

3.3. Relocation

A last problem arises when dealing with components: because the component code is relocatable, the actual base address of the component is different according to the application it is linked in. For some cases, like the instruction cache or the branch prediction analysis, the base address has an effect that must be taken into account. For example, in the instruction cache, the program addresses have an influence on the occupied cache line and blocks.

To prevent any problem, two mechanisms (possibly combined) can be used. First, the partial results may contain information that restricts the acceptable base address of the component. These restrictions ensures that the base address is chosen such that the computed partial results remains usable to

describe the WCET contribution of the component at composition time. This constraint is illustrated with the instruction cache analysis, where we impose restrictions on the component base address based on cache block alignment.

Second, if the analysis is concerned with the component base address, the *transfer* and *summary* functions must take the base address as arguments, that is, the *transfer* is defined such that $state^{after} = transfer(base,\ state^{before})$. This can be seen as a general case of the relative *transfer* and *summary* functions discussed in 2.1

3.4. Contribution to the ILP System: Minimization

In previous paragraphs, we have seen that the *summary* function takes a calling context and returns a configuration of the parameters used to instantiate the parametrized ILP system. Then, a parametrized ILP system must be returned as a part of the component partial result. Obviously, this component ILP system can be computed as if we were processing a standalone program: build structural constraints, and then build constraints related to various effects. The only difference is that we have to take care of context-dependent parts and introduce parameters accordingly. For example, the cache-related constraint could be built traditionally for all known categories, while leaving conditional rules for BB having context-dependant categories. However, this approach is not very efficient: the resulting parametrized sub-system, once instantiated and merged with the main system, gives an ILP system as large as the system of the monolithic analysis.

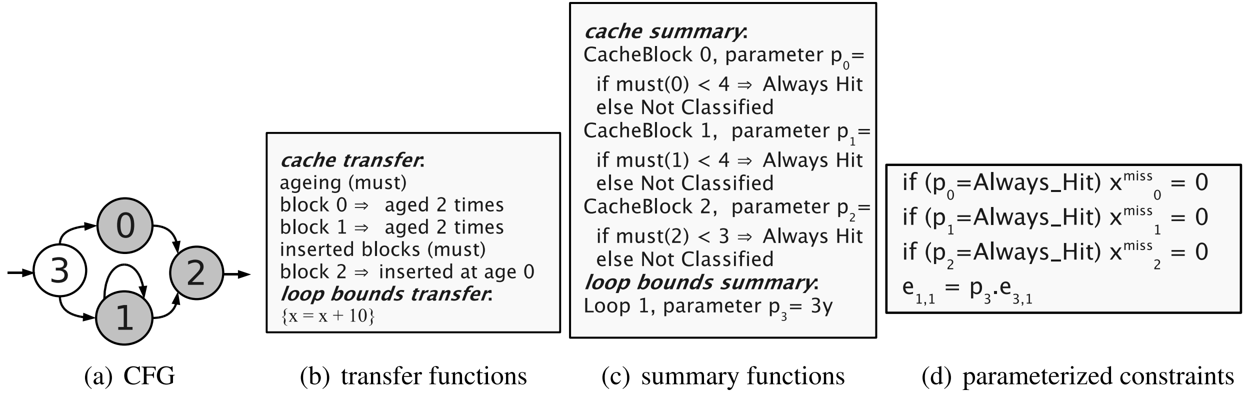
Yet, one may observe that some parts of the ILP system of a component entry point does not depend on the call context. Pre-computing these constant parts of the ILP would allow to minimize the component parametrized ILP sub-system and to speed up the WCET computation at assembly time. We have proposed a method to achieve this in [3] by CFG partitioning. It uses the concept of *Program Structure Tree* (PST) and *Single-Entry Single-Exit (*SESE*) regions* defined in [13]. We have shown that, if a SESE region is context-independent (*feasible region*), its WCET can be computed separately, and the region can be replaced by a single edge whose cost is the WCET of the region. This can be used to minimize the component ILP sub-system by pre-computing context-independent SESE regions in the component and replacing them by constant-cost edges.

To pre-compute a SESE region, two conditions must be verified: (1) the region must be a *feasible region* (that is, the ILP system restricted to the region must not depend on variables defined out of the region), and (2) the region WCET must not depend on the calling context of the component entry point function.

3.5. Example

Figure 4(a) shows an example of a component CFG where gray blocks are in the currently-analysed cache line (line 0), while the white block is in a different line. In addition, the bound of the loop at block 1 depends on an input argument named y, and there is a global variable x which is incremented by 10 due to the call of the component. The partial analysis of this component produces the transfer functions, the summary functions, and the parametrized ILP system in 4(b), 4(c) and 4(d).

The effect of the component on the cache for the Must ACS is detailed (with the ageing of each block and the list of inserted blocks with their associated ages) in transfer functions of 4(b). The effect of the component on the global variable x is also expressed. The summary function in figure 4(c) defines



(a) CFG (b) transfer functions (c) summary functions (d) parameterized constraints

Figure 4. Example

4 parameters, from p_0 to p_3. The parameters p_0, $p_1$and p_2 represents the possible categorizations for blocks 0, 1, and 2. The parameter p_3 represents the bounds of the loop, depending on the input argument y. The figure 4(d) shows the parametrized part of the ILP system (constants parts are not shown for the sake of brevity). There is a conditional constraint for each cache parameter: for each Always-Hit block, we generate a constraints setting the x_{miss} to 0 and the last constraint express the loop bounds in a parametrized way, depending on p_3.

4. Implementation

In this section, we present the implementation of our component approach with OTAWA [6].

4.1. An Exchange Format for Partial Results

A common format is required so that each component-aware analyzer can communicate with each other. Wehave chosen XML as it maximizes extensibility and flexibility. The format must sum up the various data, presented in the previous sections, that must be included in the partial result as a tree of XML elements and attributes. While some information items are generic and can be specified precisely in the format, other ones are analysis-specific and are handled as an opaque XML content.

For each component entry point, the partial result needs to include two main items: (1) for each analysis, the *transfer* and *summary* functions data and (2) for all analyses, the parametrized ILP sub-system representing the component WCET contribution (this system includes contributions from various analyzers, but the result is merged).

The *component* tag contains a collection of entry points represented by the *function* element. Each one matches an entry point, corresponding to an actual function in the binary file of the component and contains the associated partial result. The functions that are not entry points should not appear in this list: they are taken into account in the partial results of entry points calling them. The *transfer* and *summary* functions data should be represented respectively by *transfer* and *summary* XML elements, children of *function* elements. although their content is analysis-specific, XML allows us to process them easily as opaque data.

The ILP system, is stored in *function* as a *system* element. The *entry* attribute specifies the variable which represents the execution count of the entry BB. It is used to merge the component ILP system

```
<component name="comp"> <function name="funct"> <analysis type="instcache">
    <transfer>
      <damage block="0" aging="2" /> <damage block="1" aging="2" >
      <insert block="2" age="0" />
    </transfer>
    <summary> <lblock cacheblockid="1" cond_category="..." /> ... </summary>
  </analysis> <system entry="x1">
    <objective type="max" const="0">
      <term var="x1" coef="12" /> <term var="xmiss_1" coef="32" /> ...
    </objective>
    <constraint op="EQ" const="1"> <term var="x1" coef="1"/> </constraint>
    <switch param="p0">
      <case value="always_hit">
        <constraint op="EQ" const="0" > <term var="xmiss_0" coef="1"/> </constraint>
      </case>
    </switch> <constraint op="LE" const="0" >
       <term var="e_1_1" coef="1"/> <term var="e_3_1" param="p3" coef="-1"/>

    </constraint> ... </system> </function> </component>
```

Figure 5. XML format example

with the main ILP system since the WCET contribution of the component is proportional to this variable. Inside the system, the objective function part is represented by an *objective* element with *const* attribute representing the constant part. Constraints are represented by *constraint* elements and contains *op* and *const* attributes to determine the comparator type and constant.

Both the objective function and the constraints contains terms represented as *term* elements with *coef*, *var* and *param* (in case of parameter dependency) attributes. If the *var* attribute is omitted, the term is considered as a constant. Notice that *var* and *param* can not be both omitted, otherwise the term is constant and must be added to *const* attribute. The final coefficient of a term is obtained by multiplying, if any, the *coef* and the *param* actual value providing a constant ensuring that the objective function or the constraint remains affine.

While parameters in terms allows to modify coefficients, they are also used to conditionally define some constraints with *if* and *switch* elements. The *if* condition is expressed using *param, const,* and *op* attributes and embeds a *then* element and (optionally) an *else* element. The *switch* uses the tested parameter in the *param* attribute, and conditional constraints are put in *case* children elements. Each *case* elements corresponds to a parameter value (specified by *value* attribute). As an example, a part of the XML exchange file that would be generated for the component example discussed in section 3.5 is shown in figure 5. It is not complete, but contains the transfer function informations for the cache, and the parametrized part of the ILP system.

4.2. Experimentation

To validate our approach, we have first tested it with the instruction cache partial analysis presented in 2.1. The analysis were done by OTAWA [6], 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. The measurements have been done on a small subset of Mälardalen benchmarks with the *lp_solve* ILP solver. The small number of the selected benchmarks is due to the fact that (1) some benchmarks do not contain any function (except *main*) and (2) the provided functions are too small to make them interesting component entry points.

While the component partial analysis is meant to allow COTS usage, a nice side-effect is that it makes the analysis faster because (1) the sections of the partial result are reused when the entry point is

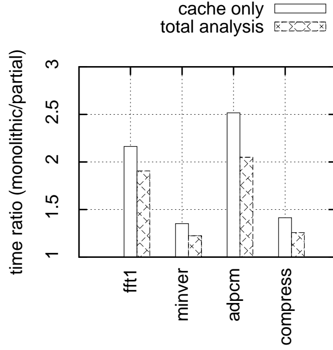


Figure 6. time gain

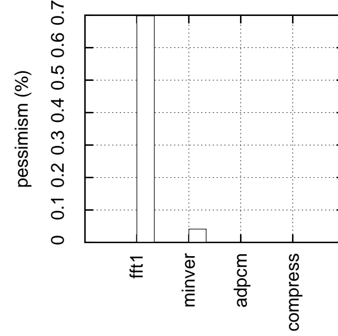


Figure 7. pessimism measurement

called more than once (due to regions and ILP system minimization) and (2) the WCET computation is usually of exponential complexity, so it is faster to analyse components separately than to analyse the whole program. We have measured this speed increase in figure 6: for each benchmark we display the ratio between the time with partial analysis and without. There is a measurement for the whole WCET computation (including the ILP solving), and one for the cache analysis time only. On average, the partial analysis is 1.6 times faster (even though our component is called few times in our small benchmarks). When accounting only for the cache analysis time, the average time gain ratio is of 1.86 (applying transfer and summary for the cache is immediate, while the parametrized ILP system instantiation requires handling regions that could not be pre-computed).

A drawback of our method is that the *transfer* and *summary* are not guaranteed to be exact. They must at least be conservative (for example, for the cache transfer function, the real ACS_{must}^{after} that would have been computed by a real analysis of the component must be *less or equal* than the one computed by $transfer(ACS_{must}^{before})$). This may lead to pessimism, which we have measured: in the figure 7, for each benchmark we display the ratio between WCET with partial analysis and without. Fortunately, we detect that, on average, the WCET increase is only 0.18 % (the pessimism is caused by the *transfer* and *summary* functions because the region optimization method is exact).

In addition to this experiments, we have tested our loop bound partial analysis with oRange [10] on the following Mälardalen benchmarks providing enough interesting components with loops: *adpcm*, *cnt, crc, duff, expint, jfdctint, ludcmp*. On all the performed tests, the loop bound estimation showed the same results with and without the partial analysis. Furthermore, the few big-enough test (*adpcm*) exhibited a 1.5 times faster partial analysis (other tests were too small and therefore too quickly processed by oRange to deduce any valuable performance information).

5. Related work

In [5], S. Bygde and B. Lisper proposes an approach for parametric WCET computation. This method addresses the fact that the WCET of a program depends on the input variables and arguments. Some parameters, introduced in the IPET system, are expressed in terms of the input variables. Then, the method for parametric ILP resolution of [11] is applied to get a parametric, tree-based WCET, with nodes representing tests on parameters and leaves representing numerical WCET values.

In [17], F. Mueller extends the method defined in [15, 16] to a program made up of multiple modules. First, a module-level analysis, consisting of four analyses for different possible contexts (scopes),

is done independently for each module, resulting in temporary categories. Next, a compositional analysis is performed on the whole program to adjust the categories according to the call context. Mueller's approach is focused on the performance gain resulting from the partial analysis, but does not appear to be designed to handle COTS, since the compositional step needs to visit each component CFG. Moreover, his approach needs more analyses passes than ours, and is bound to direct-mapped instruction caches.

In [18], a method is proposed to perform component-wise instruction-cache behavior prediction. It adapts [1] analysis to an executable linked from multiple components by analyzing independently the object files (producing partial results), and composing the partial results. Two main aspects of the problem are addressed: (1) the absolute base address of the component is not known during the analysis, so the linker is instructed to put the component into a memory location that is equivalent (from the cache-behavior point of view) to the memory location that was used in the analysis; and (2) when a component calls a function in another component, the called function has an influence on the cache state, and on the ACS of the caller. To handle this problem, the paper defines, for each called component, a cache damage function which performs the same role as the cache transfer function described in 2.1 We have extended this approach by adding the *Persistence* analysis.

One may observe that the two last papers provide a working approach for components but they are bound to the instruction cache analysis. In the opposite, our approach proposes a generic framework that supports several types of analyses.

6. Conclusion

In this paper we have proposed a generic approach to handle blackbox component partial analysis, which enables us to compute partial WCET for components, and then to reuse this result in the WCET computation of a main application. By surveying some existing IPET-based partial analysis (the instruction cache, and the loop bound estimation), we have generalized this approach to a generic framework integrating partial analyses and contribution to the WCET ILP system, and we have defined a common partial result exchange format in XML, extensible and adaptable to other WCET-related analyses.

While this approach is primarily designed to handle COTS, tests with OTAWA and oRange have shown that (1) due to exponential complexity of WCET computation and (2) to the partial result reuse, we also gain a significant analysis time without sacrificing precision. Adaptation of others partial analysis in OTAWA (like branch prediction) is underway. It would be also interesting to continue experimentations on bigger benchmarks.

In addition, we would like to fix one drawback of our approach: it needs ad-hoc derivation of partial analyses from existing, non-partial ones. In the future, for some analysis based on abstract interpretation, we want to survey methods to help to or to derive automatically *transfer* and *summary* functions from a description of the analysis. At least, it may be valuable to provide a framework to help writing partial analyses by automating the more used parts.

References

[1] ALT, M., FERDINAND, C., MARTIN, F., AND WILHELM, R. Cache behavior prediction by abstract interpretation. In *SAS'96* (1996), Springer-Verlag.

[2] BALLABRIGA, C., AND CASSÉ, H. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *ECRTS'08* (2008).

[3] BALLABRIGA, C., AND CASSÉ, H. Improving the WCET computation time by IPET using control flow graph partitioning. In *International Workshop on Worst-Case Execution Time Analysis (WCET), Prague* (juillet 2008).

[4] BALLABRIGA, C., CASSÉ, H., AND SAINRAT, P. An improved approach for set-associative instruction cache partial analysis. In *SAC'08* (March 2008).

[5] BYGDE, S., AND LISPER, B. Towards an automatic parametric wcet analysis. In *WCET'08.*

[6] CASSÉ, H., AND SAINRAT, P. OTAWA, a framework for experimenting WCET computations. In *ERTS'05* (2005).

[7] COLIN, A., AND PUAUT, I. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems 18*, 2-3 (2000).

[8] COUSOT, P., AND COUSOT, R. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming* (1976), Dunod.

[9] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT* (1977).

[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 RTCSA'08* (August 2008).

[11] FEAUTRIER, P. Parametric integer programming. *RAIRO Recherche Opérationnelle 22*, 3 (1988).

[12] FERDINAND, C., MARTIN, F., AND WILHELM, R. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN workshop on language, compiler and tool support for real-time systems* (1997).

[13] JOHNSON, R., PEARSON, D., AND PINGALI, K. The program structure tree: Computing control regions in linear time. In *SIGPLAN PLDI'94* (1994).

[14] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems* (1995).

[15] MUELLER, F. Timing analysis for instruction caches. *Real-Time Systems* (May 2000).

[16] MUELLER, F., AND WHALLEY, D. Fast instruction cache analysis via static cache simulation. *TR 94042, Dept. of CS, Florida State University* (April 1994).

[17] PATIL, K., SETH, K., AND MUELLER, F. Compositional static instruction cache simulation. *SIGPLAN Not. 39*, 7 (2004).

[18] RAKIB, A., PARSHIN, O., THESING, S., AND WILHELM, R. Component-wise instruction-cache behavior prediction. In *Automated Technology for Verification and Analysis* (2004).

ALF – A LANGUAGE FOR WCET FLOW ANALYSIS

Jan Gustafsson[1], Andreas Ermedahl[1], Björn Lisper[1], Christer Sandberg[1], and Linus Källberg[1]

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 the flow analysis, which derives bounds on the number of times different code entities can be executed. Examples of flow information derived by a flow analysis are loop bounds and infeasible paths.
Flow analysis can be performed on source code, intermediate code, or binary code: for the latter, there is a proliferation of instruction sets. Thus, flow analysis must deal with many code formats. However, the basic flow analysis techniques are more or less the same regardless of the code format. Thus, an interesting option is to define a common code format for flow analysis, which also allows for easy translation from the other formats. Flow analyses for this common format will then be portable, in principle supporting all types of code formats which can be translated to this format. Further, a common format simplifies the development of flow analyses, since only one specific code format needs to be targeted.
This paper presents such a common code format, the ALF language (ARTIST2 Language for WCET Flow Analysis).

1. Introduction

Bounding the Worst-Case Execution Time (WCET) is crucial when 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, as well as the program's possible execution flows, needs to be derived. The latter includes information about the maximum number of times loops are iterated, infeasible paths, etc. The goal of a *flow analysis* is to calculate such flow information as automatically as possible. A precise flow analysis is of great importance for calculating a tight WCET estimate [18].

[1] School of Innovation, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden.
{jan.gustafsson,andreas.ermedahl,bjorn.lisper,christer.sandberg,
linus.kallberg}@mdh.se

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2279

The input to the flow analysis is a representation of the program to be analysed as, e.g., binary code, intermediate code, or source code. These alternatives have different pros and cons:

Binary code. This is the code actually being run on the processor, and thus the code for which the flow information is relevant. Analyzing the binary code therefore ensures that the correct flow information is found. However, information available in the source code may be lost in the binary, which can lead to a less precise flow analysis. To do a good job, an analysis tool will have to reconstruct the high-level program structure such as the control flow graph, function calls and returns, etc. The WCET analysis tools aiT [2] and Bound-T [7] analyze binary code, and include both a type of instruction decoding- and program-graph reconstruction phase [6, 19].

Source code. Source code can typically be analyzed more precisely, since there is much more information present. On the other hand, compiler optimizations can cause the control structure of the generated binary code to be different from the one of the source code. Therefore, to be used for WCET analysis the flow information must be transformed accordingly [10]. This requires compiler support, which current production compilers do not offer. On the other hand, source-code analysis can be used for other purposes such as deriving and presenting program flow information to the developer. Thus, flow analysis of source code is definitely of interest. Typically, when source code is analysed, the code is often translated to some type of intermediate code which is close to the source code. An example of a WCET analysis tool which works on source code level is TuBound [16].

Intermediate code. This kind of code is typically used by compilers, for the purpose of optimizing transformations, before the binary code is generated. It is often quite rich in information. The intermediate code generated by the parser is usually more or less isomorphic to the source code, while after optimizations it typically has a program flow which is close to the flow in the generated binary code. These properties make intermediate code an interesting candidate for flow analysis, since it can be analyzed for flow properties of both source and binary code. On the downside, the analysis becomes dependent on a certain compiler: code generated by another compiler cannot be analyzed. The current version of the WCET analysis tool SWEET analyzes the intermediate format of the NIC research compiler [4].

ALF is developed with flexibility in mind. The idea behind ALF is to have a generic language for WCET flow analysis, which can be generated from all of the program representations mentioned above. In this paper we describe ALF, its intended use, and some current projects where ALF will be used. For a complete description of ALF, see the language specification [3].

The rest of the paper is organized as follows: Section 2 describes the ALF language. Section 3 presents the intended uses of ALF, and in Section 4 we describe some of the current projects where ALF is being used. Section 5 presents some related work, and in Section 6 we draw some conclusions and discuss future work.

2. ALF (ARTIST2 Language for WCET Flow Analysis)

ALF is a language to be used for flow analysis for WCET calculation. It is an intermediate level language which is designed for analyzability rather than code generation. It is furthermore designed

to represent code on source-, intermediate- and binary level (linked as well as unlinked) through relatively direct translations, which maintain the information present in the original code needed to perform a precise flow analysis.

ALF is basically a sequential imperative language. Unlike many intermediate formats, ALF has a fully textual representation: it can thus be seen as an ordinary programming language, although it is intended to be generated by tools rather than written by hand.

2.1. Syntax

ALF has a Lisp/Erlang-like syntax, to make it easy to parse and read. This syntax uses prefix notation as in Lisp, but with curly brackets "{", "}" as parentheses as in Erlang. An example is

```
{ dec_unsigned 32 2 }
```

which denotes the unsigned 32-bit constant 2.

2.2. Memory Model

ALF's memory model distinguishes between program and data addresses. It is essentially a memory model for relocatable, unlinked code. Program and data addresses both have a symbolic base address, and a numerical offset. Program addresses are called *labels*. The address spaces for code and data are disjoint. Only data can be modified: thus, self-modifying programs cannot be modelled in ALF in a direct way.

2.3. Program Model

ALF's program model is quite high-level, and similar to C. An ALF program is a sequence of declarations, and its executable code is divided into a number of function declarations. Within each function, the program is a linear sequence of statements, with execution normally flowing from one statement to the next. Statements may be tagged with labels. ALF has jumps, which can go to dynamically calculated labels: this can be used to represent program control in low-level code. In addition ALF also has structured function calls, which are useful when representing high-level code. See further Sections 2.6 and 2.7

A function named "main" will be executed when an ALF program runs. ALF programs without a main function cannot be run, but may still be analyzed.

2.4. Data Model

ALF's data memory is divided into *frames*. Each frame has a symbolic base pointer (a *frameref*) and a *size*. A data address pointing into a frame is formed from the frameref of the frame and an offset. The offset is a natural number in the *least addressable unit* (LAU) of the ALF program. The LAU is always declared: typically it is set to a byte (8 bits).

Frames can be either statically or dynamically allocated. Statically allocated memory is explicitly declared. There are two ways to allocate memory dynamically:

- As local data in so-called *scopes*. This kind of local data is declared like statically allocated data, but in the context of a scope rather than globally. It would typically be allocated on a stack.
- By calling a special function "`dyn_alloc`" (similar to `malloc` in C) that returns a frameref to a newly allocated frame. This kind of data would typically be allocated on a heap.

Semantically, framerefs are actually pairs (i, n) where i is an identifier and n a natural number. For framerefs pointing to statically allocated frames, n is zero. For frames dynamically allocated with `dyn_alloc`, each new allocation returns a new frameref with n increased by one. This semantics makes it possible to perform static analysis of dynamically allocated memory, like bounding the maximal number of allocations to a certain data area.

ALF's data memory model can be used both for high-level code, intermediate formats, and binaries, like in the following:

- "High-level": allocate one frame per high-level data structure in the program (struct, array, ...)
- "Intermediate-level": use one frameref identifier for each stack, heap etc. in the runtime system, in order to model it by a potentially unbounded "array" of frames (one for each object stored in the data structure). Use one frame for each possible register. If the intermediate model has an infinite register bank, then use one identifier to model the bank (again one frame per register).
- Binaries: allocate one frame per register, and a single frame for the whole main memory.

2.5. Values and Operators

Values can be:

- Numerical values: signed/unsigned integers, floats, etc.,
- Framerefs,
- *Label references* (lrefs), which are symbolic base addresses to code addresses (labels),
- Data addresses (f, o), where f is a frameref and o is an offset (natural number), or
- Code addresses (labels) (f, n), where f is an lref and n a natural number.

There is a special value `undefined`. This provides a fallback in situations where an ALF-producing tool, e.g., a binary-to-ALF translator, cannot translate a piece of the binary code into sensible ALF.

Each value in ALF has a *size*, which is the number of bits needed for storing it. ALF has integers of unbounded size: these are used mainly when translating language constructs into ALF that are not easily modelled with ALF's predefined operators. All other values have finite size, and they are *storable*, meaning that they can be stored in memory. Storable values are further subdivided into *bitstring* values, which have a unique bitstring representation, and *symbolic* values. The latter include

address values, since they have symbolic base addresses. Only a limited number of operations are allowed on symbolic values.

Bitstring values are primarily numeric values. A bitstring value can be numerically interpreted in various ways depending on its use: as signed, unsigned, floating-point, etc. This is a complication when performing a value analysis of ALF, since this analysis must take the different possible interpretations into account. A *soft type system* [20] for ALF, which can detect how values are used and restrict their possible interpretations, is under development [3].

ALF has an explicit syntax for constant values, which includes the size and the intended interpretation. This is intended to aid the analysis of ALF programs.

Operators in ALF are of different kinds. The most important ones are the operators on *data of limited size*. These operators mostly model the functional semantics of common machine instructions (arithmetic and bitwise logical operations, shifts, comparisons, etc). ALF also has operators on *data of unbounded size*. These are "mathematical" operations, typically on integers: for instance, all "usual" arithmetic/relational operators have versions for unbounded integers. They are intended to be used to model the functional semantics for instructions whose results cannot be expressed directly using the finite-size-data operators. An example is the settings of different flags after having carried out certain operations, where ALF sometimes does not provide a direct operator.

Furthermore, ALF has a few operators on *bitstrings*, like bitstring concatenation. They are intended to model the semantics on a bitstring level, which is appropriate for different operations involving masking etc. There is a variety of such operations in different instruction sets, and it would be hard to provide direct operators for them all. ALF also has a *conditional*, and a *conversion function* from bitstrings to natural numbers. These functions are useful when defining the functional semantics of instructions.

Finally, ALF has a *load* operator that reads data from a memory address, and the aforementioned `dyn_alloc` function that allocates a new frame.

All these operators, except `dyn_alloc`, are side-effect free. This simplifies the analysis of ALF programs. Each operator takes one or several *size arguments*, which give the size(s) of the operands, and the result. ALF is very explicit about the sizes of operands and operators, in order to simplify analysis and make the semantics exact.

2.6. Statements

ALF has a number of statements. These are similar to statements in high-level languages in that they typically take full expressions as arguments, rather than register values. The most important are the following:

- A concurrent assignment (`store`), which stores a list of values to a list of addresses. Both values and addresses are dynamically evaluated from expressions.
- A multiway jump (`switch`), which computes a numerical value and then jumps to a dynamically computed address depending on the numerical value. For convenience, ALF also has an unconditional `jump` statement.

- Function call and return statements. The call statement takes three arguments: the function to be called (a code address), a list of actual arguments, and a list of addresses where the return values are to be stored upon return from the called function. Correspondingly, the return statement takes a list of expressions as arguments, whose values are returned. The address specifying the function to be called can be dynamically computed.

2.7. Functions

ALF has procedures, or *functions*. A function has a name and a list of formal arguments, which are frames. The body of a function is a scope, which means that a function also can have local variables. The formal arguments are similar to locally declared variables in scopes, with the difference that they are initialized with the values of the actual arguments when the function is called.

As explained in Section 2.6, a function is exited by executing a return statement. A function is also exited when the end of its body is reached, but the result parameters will then be undefined after the return of the call. Functions can be recursive.

Functions can only be declared at a single, global level, and a declared function is visible in the whole ALF program. ALF has lexical (static) scoping: locally defined variables, or formal arguments, take precedence over globally defined entities with the same name. This is similar to C.

2.8. The Semantics of ALF

ALF is an imperative language, with a relatively standard semantics based on state transitions. The state is comprised of the contents in data memory, a program counter (PC) holding the (possibly implicit) label of the current statement to be executed, and some representation of the stacked environments for (possibly recursive) function calls.

2.9. An Example of ALF Code

The following C code:

```
if(x > y) z = 42;
```

can be translated into the ALF code below:

```
{ switch { s_le 32 { load 32 { addr 32 { fref 32 x } { dec_unsigned 32 0 } } }
                   { load 32 { addr 32 { fref 32 y } { dec_unsigned 32 0 } } } }
         { target { dec_unsigned 1 1 }
                  { label 32 { lref 32 exit } { dec_unsigned 32 0 } } } }
{ store { addr 32 { fref 32 z } { dec_unsigned 32 0 } }
        with { dec_signed 32 42 } }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } }
```

The if statement is translated into a switch statement jumping to the exit label if the (negated) test becomes true (returns one). The test uses the s_le operator (signed less-than or equal), taking 32 bit arguments and returning a single bit (unsigned, size one). Each variable is represented by a frame of size 32 bits.

3. Flow Analysis Using ALF

ALF will be able to offer a high degree of flexibility. Provided that translators are available, the input can be selected from a wide set of sources, like linked binaries, source code, and compiler intermediate formats: see Section 4.3 for some available translators. The flow analysis is then performed on the ALF code.

If a single, standardized format like ALF is used, then it is natural to implement different flow analyses for this format. This facilitates direct comparisons between different analysis methods. Also, since different analyses are likely to be the most effective ones for different kinds of translated input formats, the analysis of heterogeneous programs is facilitated. This can be useful in situations where parts of the analysed program are available as, say, source code, while other parts, like library routines, are available only as binaries. Also, different parts of the program with different characteristics may require different flow analysis methods, e.g., input dependent functions may require a parametric analysis [12]. Flow analysis methods can be used as "plug-ins" and even share results with other methods. Several flow analysis methods can be used in parallel and the best results can be selected.

ALF itself does not carry flow data. The flow analysis results must be mapped back as flow constraints on the code from which the ALF code was generated. An ALF generator can use conventions for generating label names in order to facilitate this mapping back to program points in the original code.

As a future general format for flow constraints, the *Program Flow Fact* (PFF) format is being developed to accompany ALF. PFF will be used to express the dynamic execution properties of a program, in terms of constraints on the number of times different program entities can be executed in different program contexts.

4. Current Projects Using ALF

4.1. ALL-TIMES

ALL-TIMES [5] is a medium-scale focused-research project within the 7th Framework Programme. The overall goal of ALL-TIMES is to integrate European timing analysis technology. The project is concerned with embedded systems that are subject to safety, availability, reliability, and performance requirements. These requirements often relate to correct timing, notably in the automotive and aerospace areas. Consequently, the need for appropriate timing analysis methods and tools is growing rapidly.

ALL-TIMES will enable interoperability of the various tools from leading commercial vendors and universities alike, and develop integrated tool chains using open tool frameworks and interfaces. The different types of connections between the tools are:

- provision of data: one tool provides some of its analysis data to a another tool
- provision of component: the component of one tool is provided to another partner for integration in the other tool
- sharing of data: this is a bidirectional connection where similar data is computed by both tools and the connector allows to exchange that information in both directions

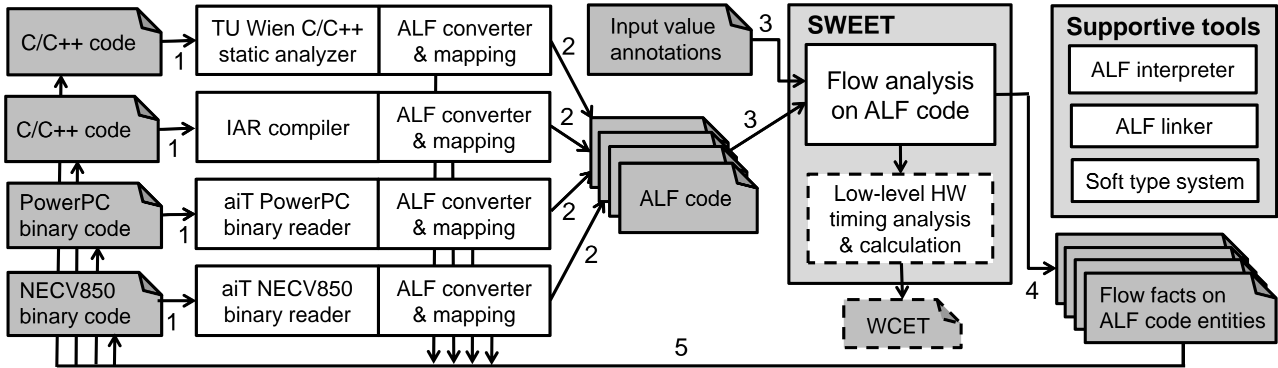


Figure 1. The use of ALF with the SWEET tool

- combination of features: the foreseen connection allows to combine features from different tools in both directions (either by exchanging data or components)

ALF is being developed within the ALL-TIMES project, where it will contribute to the interoperability between WCET tools.

4.2. ALF and SWEET

ALF will be used as input format for the WCET analysis tool SWEET (SWEdish Execution Time tool), which is a prototype WCET analysis tool developed at Mälardalen University [13].

Figure 1 describes the intended use of ALF in conjunction with SWEET. Step 1 represents the reading of the input program code, represented in different formats and levels. Step 2 describes the output in the generic ALF format. Step 3 shows the inputs to the flow analysis, which (in Step 4) outputs the results as flow constraints (flow facts) on ALF code entities. Using the mapping information created earlier, these flow facts can be mapped back to the inputs formats (Step 5). The four different translators to ALF are described shortly below.

4.3. Translators to ALF

A number of translators to ALF code are currently being developed. All three types of sources will be covered; source code, intermediate code, and binary code.

C/C++ source code to ALF translator. This C/C++ to ALF translator is developed within the ALL-TIMES project. It is based on the SATiRE framework [17], developed by the Compilers and Languages Group of the Institute of Computer Languages at TU Vienna. SATIrE integrates components of different program analysis tools and offers an infrastructure for building new code analysis tools. The C to ALF translator is available on the MELMAC web site [14]. The mapping of flow constraints back to the source code will use the Analysis Results Annotation Language (ARAL) format, which is a part of the SATiRE framework.

IAR intermediate code to ALF translator. This translator converts a proprietary intermediate code format, used by the embedded systems compiler vendor IAR Systems AB [8], into ALF.

Binary code to ALF translators. Two binary translators are currently under development at Mälardalen University, within the ALL-TIMES project. They translate CRL2 to ALF. CRL2 is a data format maintained by AbsInt GmbH [1], which is used by the aiT WCET analysis tool. CRL2 is mainly used to represent various types of assembler code formats with associated analysis results. By using CRL2 as source, the existing binary readers of aiT can be used to decode the binaries.

The first translator[1] translates CRL2's representation of NECV850E assembler code to ALF, and the second translator[2] converts PowerPC code into ALF. To allow SWEET's analysis results to be given back to CRL2, a mapping between CRL2's and ALF's different code and data constructs will be maintained. For both translators the generated flow constraints will be transferred to aiT using the AIS format, which is the annotation format used by aiT.

4.4. ALF interpreter

An ALF interpreter is currently being developed at Mälardalen University[3]. The interpreter will be used for debugging ALF programs (or, indirectly, translators generating ALF), and other purposes.

4.5. ALF file linking

An embedded systems project may involve a variety of code sources, including code generated from modelling tools, C or C++ code, or even assembler. To form a binary executable program, the source files are compiled and linked together with other object code files and libraries [11]. The latter are often not available in source code format [15].

To handle this issue ALF must support the "linking" of several ALF files into one single ALF file. Since ALF is supposed to support reverse engineering of binaries and object code, it cannot have an advanced module system with different namespaces. Instead ALF provides, similar to most object code formats [11], a set of exported symbols, i.e., `lrefs` or `frefs` visible to other ALF files, as well as a set of symbols that must be imported, i.e., `lrefs` or `frefs` used but not declared in the ALF file.

Forthcoming work includes the development of an *ALF linker*, i.e., a tool able to "link" ALF files generated from many different code sources. It will here be interesting to see if it is also possible to "link" (maybe partial) flow analysis results for individual ALF files to form a flow analysis result valid for the whole program.

5. Related Work

Generic intermediate languages have been used for other purposes. One example is the hardware description language TDL [9] which was designed with the goal to generate machine-dependent post-pass optimizers and analyzers from a concise specification of the target processor. TDL provides a generic modelling of irregular hardware constraints that are typical for many embedded processors. The part of TDL that is used to describe the semantics of instruction sets resembles ALF to some degree.

[1] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0875
[2] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0917
[3] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0830

6. Conclusions and Future Work

ALF can be seen as a step towards an open framework where flow analyses can be available and freely used among several WCET analysis research groups and tool vendors. This framework will ease the comparison of different flow analysis methods, and it will facilitate the future development of powerful flow analysis methods for WCET analysis tools.

There may be other uses for ALF than supporting flow analysis for WCET analysis tools. For instance, ALF can be used as a generic representation for different binary formats. Thus, very generic tools for analysis, manipulation, and reverse engineering of binary code may be possible to build using ALF. Possible examples include generic binary readers that reconstruct control flow graphs, and tools that can reconstruct arithmetics for long operators implemented using instruction sets for shorter operators. The latter can be very useful when performing flow analysis for binaries compiled for small embedded processors, where the original arithmetics in the source code must be implemented using an instruction set for short operators.

It would be simple to define an XML syntax for ALF. We might do so in the future, in order to facilitate the use of standard tools for handling XML. Future work also includes further development of the PFF format for flow facts and the format for input value annotations.

Acknowledgment

This research was supported by the KK-foundation through grant 2005/0271, and the ALL-TIMES FP7 project, grant agreement no. 215068.

References

[1] ABSINT. aiT tool homepage, 2008. `www.absint.com/ait`.

[2] FERDINAND, C., HECKMANN, R., AND FRANZEN, B. Static memory and timing analysis of embedded systems code. In *3rd European Symposium on Verification and Validation of Software Systems (VVSS'07), Eindhoven, The Netherlands* (Mar. 2007), no. TUE Computer Science Reports 07-04, pp. 153–163.

[3] GUSTAFSSON, J., ERMEDAHL, A., AND LISPER, B. ALF (ARTIST2 Language for Flow Analysis specification. Tech. rep., Mälardalen University, Västerås, Sweden, Jan. 2009.

[4] GUSTAFSSON, J., LISPER, B., SANDBERG, C., AND BERMUDO, N. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)* (Jan. 2003).

[5] GUSTAFSSON, J., LISPER, B., SCHORDAN, M., FERDINAND, C., GLIWA, P., JERSAK, M., AND BERNAT, G. ALL-TIMES - a European project on integrating timing technology. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods (ISOLA'08)* (Porto Sani, Greece, Oct. 2008), vol. 17 of *CCIS*, Springer, pp. 445–459.

[6] HOLSTI, N. Analysing switch-case tables by partial evaluation. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)* (2007).

[7] HOLSTI, N., AND SAARINEN, S. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)* (2002).

[8] IAR Systems homepage.
URL: `http://www.iar.com`.

[9] KÄSTNER, D. TDL: A hardware description language for retargetable postpass optimizations and analyses. In *Proc. 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, Lecture Notes in Computer Science (LNCS) 2830. Springer-Verlag, 2003.

[10] KIRNER, R. *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.

[11] LEVINE, J. *Linkers and Loaders*. Morgan Kaufmann, 2000. ISBN 1-55860-496-0.

[12] LISPER, B. Fully automatic, parametric worst-case execution time analysis. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)* (Porto, July 2003), J. Gustafsson, Ed., pp. 77–80.

[13] MÄLARDALEN UNIVERSITY. WCET project homepage, 2008. `www.mrtc.mdh.se/projects/wcet`.

[14] MELMAC. MELMAC homepage, 2009.
`http://www.complang.tuwien.ac.at/gergo/melmac`.

[15] MONTAG, P., GOERZIG, S., AND LEVI, P. Challenges of timing verification tools in the automotive domain. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[16] PRANTL, A., SCHORDAN, M., AND KNOOP, J. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)* (Prague, Czech Republic, July 2008), R. Kirner, Ed., pp. 141–148.

[17] SATIRE. SATIrE homepage, 2009.
`http://www.complang.tuwien.ac.at/markus/satire`.

[18] SEHLBERG, D., ERMEDAHL, A., GUSTAFSSON, J., LISPER, B., AND WIEGRATZ, S. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[19] THEILING, H. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, University of Saarland, 2002.

[20] WRIGHT, A. K., AND CARTWRIGHT, R. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst. 19*, 1 (Jan. 1997), 87–152.

CACHE-RELATED PREEMPTION DELAY COMPUTATION FOR SET-ASSOCIATIVE CACHES

PITFALLS AND SOLUTIONS[1]

Claire Burguière, Jan Reineke, Sebastian Altmeyer[2]

Abstract
In preemptive real-time systems, scheduling analyses need—in addition to the worst-case execution time—the context-switch cost. In case of preemption, the preempted and the preempting task may interfere on the cache memory. These interferences lead to additional reloads in the preempted task. The delay due to these reloads is referred to as the cache-related preemption delay (CRPD). The CRPD constitutes a large part of the context-switch cost. In this article, we focus on the computation of upper bounds on the CRPD based on the concepts of useful cache blocks (UCBs) and evicting cache blocks (ECBs). We explain how these concepts can be used to bound the CRPD in case of direct-mapped caches. Then we consider set-associative caches with LRU*,* FIFO*, and* PLRU *replacement. We show potential pitfalls when using UCBs and ECBs to bound the CRPD in case of* LRU *and demonstrate that neither UCBs nor ECBs can be used to bound the CRPD in case of* FIFO *and* PLRU*. Finally, we sketch a new approach to circumvent these limitations by using the concept of relative competitiveness.*

1. Introduction

Preemption introduces a new dimension of complexity into worst-case execution time (WCET) analysis: The possible interference of preempting and preempted task—especially on the cache—has to be taken into account. The additional execution time due to preemption is referred to as the context-switch cost (CSC), the part of the context switch cost due to cache interferences as cache-related preemption delay (CRPD). One approach to soundly deal with cache interferences due to preemption is to totally avoid them by cache partitioning. When cache partitioning is not an option, one can distinguish two other approaches: 1) to incorporate cache interferences within WCET analysis, or 2) to perform a separate analysis of the number of additional misses due to preemption. Whereas only the first alternative is applicable to processors exhibiting timing anomalies, the second one is probably more precise but relies on "timing compositional" processors.

Lee et al. [2] laid the foundation for the separate analysis of the cache-related preemption delay by introducing the notion of *useful cache block* (UCB). A cache block of the preempted task is called useful at program point P, if it may be cached at P and if it may be reused at some program point reached from P. Memory blocks that meet both criteria may cause additional cache misses that only occur in case of preemption. Hence, an upper bound on the number of UCBs gives an upper bound

[1]This work was supported by ICT project PREDATOR in the European Community's Seventh Framework Programme, by Transregional Collaborative Research Center AVACS of the German Research Council (DFG) and by ARTIST DESIGN NoE.

[2]Compiler Design Lab, Saarland University, 66041 Saarbrücken, Germany
{burguiere,reineke,altmeyer}@cs.uni-saarland.de

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2285

on the number of additional misses and, multiplied by the cache miss penalty, an upper bound on the cache-related preemption delay.

As Tan et al. [7] and, later, Staschulat et al. [6] have shown, the CRPD bound can be reduced by taking into account the preempting task. Only those cache blocks that are actually evicted due to preemption may contribute to the CRPD. So, the preempting task is analyzed in order to derive the set of *evicting cache blocks* (ECBs) and to bound the effect of the preempting task on the useful cache blocks. These concepts have been introduced for direct-mapped and set-associative caches with LRU replacement. Although widely used in practice, other replacement policies, such as PLRU or FIFO, have not been considered so far—except for [6] which suggests to adapt the CRPD computation for LRU to PLRU.

We first review how the CRPD can be bounded for direct-mapped caches. Then, we discuss CRPD computation for set-associative caches. We show that neither the number of UCBs nor the number of ECBs can be used to bound the CRPD for set-associative caches. Even for LRU, the previously proposed combination of ECBs and UCBs may underestimate the CRPD. We provide a correct formula for LRU and sketch a new approach to CRPD computation for FIFO, PLRU, and other policies.

1.1. Background: Cache Memory

Caches are fast and small memories storing frequently used memory blocks to close the increasing performance gap between processor and main memory. They can be implemented as data, instruction or combined caches. An access to a memory block which is already in the cache is called a *cache hit*. An access to a memory block that is not cached, called a *cache miss*, causes the cache to load and store the data from the main memory.

Caches are divided into *cache lines*. *Cache lines* are the basic unit to store *memory blocks*, i.e., *line-size* l contiguous bytes of memory. The set of all memory blocks is denoted by M. The cache size s is thus given by the number of cache lines c times the line-size l. A set of n cache lines forms one *cache set*, where n is the *associativity* of the cache and determines the number of cache lines a specific memory block may reside in. The number of sets is given by c/n and the cache set that memory block b maps to is given by $b\ \mathit{mod}(c/n)$.

Special cases are direct-mapped caches ($n = 1$) and fully-associative caches ($n = c$). In the first case, each cache line forms exactly one cache set and there is exactly one position for each memory block. In the second case, all cache lines together form one cache sets and all memory blocks compete for all positions. If the associativity is higher than 1 a *replacement policy* has to decide in which cache line of the cache set a memory block is stored, and, in case all cache lines are occupied, which memory block to remove. The goal of the replacement policy is to minimize the number of cache misses.

We will investigate CRPD computation in the context of the following three widely-used policies:

- Least-Recently-Used (LRU) used in INTEL PENTIUM I and MIPS 24K/34K
- First-In, First-Out (FIFO or Round-Robin) used in MOTOROLA POWERPC 56X, INTEL XSCALE, ARM9, ARM11
- Pseudo-LRU (PLRU) used in INTEL PENTIUM II-IV and POWERPC 75X

We will explain these policies in Section 3

2. Bounding the CRPD for direct-mapped caches

The cache-related preemption delay denotes the additional execution time due to cache misses caused by a single preemption. Such cache misses occur, if the preempting task evicts cache blocks of the preempted task that otherwise would later be reused. Therefore upper bounds on the CRPD can be derived from two directions: bounding the worst-case effect on the preempted task or bounding the effect of the preempting task. Note that the schedulability analysis might have to account for multiple preemptions. In that case, it has to multiply the CRPD bound by the number of preemptions.

For the analysis of the preempted task, Lee et al. [2] introduced the concept of useful cache blocks:

Definition 1 (Useful Cache Block (UCB))
A memory block m is called a useful cache block at program point P, if

a) m may be cached at P

b) m may be reused at program point Q that may be reached from P without eviction of m on this path.

In case of preemption at program point P, only the memory blocks that a) are cached and b) will be reused, may cause additional reloads. Hence, the number of UCBs at program point P gives an upper bound on the number of additional reloads due to a preemption at P. A global bound on the CRPD of the whole task is determined by the program point with the highest number of UCBs.

The worst-case effect of the preempting task is given by the number of cache blocks the task may evict during preemption. Obviously, each memory block possibly cached during the execution of the preempting task may evict a cache block of the preempted one:

Definition 2 (Evicting Cache Blocks (ECB))
A memory block of the preempting task is called an evicting cache block, if it may be accessed during the execution of the preempting task.

Accessing an ECB in the preempting task may evict a cache block of the preempted task. Tomiyama and Dutt [8] proposed to use only the number of ECBs—in a more precise manner—to bound the CRPD. Negi et al. [3] and Tan et al. [7] proposed to combine the number of ECBs and UCBs to improve the CRPD bounds; only useful cache blocks that are actually evicted by an evicting cache block may contribute to the CRPD.

Hence, the following three formulas can be used to bound the CRPD for direct-mapped caches (where c denotes the number of cache sets):

$$\text{CRPD}_{\text{UCB}} = \text{CRT} \cdot |\{s_i \mid \exists m \in \text{UCB} : m \bmod c = s_i\}| \tag{1}$$

The CRPD is bounded by the cache reload time (CRT) times the number of sets, which at least one UCB maps to [2].

$$\text{CRPD}_{\text{ECB}} = \text{CRT} \cdot |\{s_i \mid \exists m \in \text{ECB} : m \bmod c = s_i\}| \tag{2}$$

The CRPD is bounded by the cache reload time times the number of sets, which at least one ECB maps to [8].

$$\text{CRPD}_{\text{UCB\&ECB}} = \text{CRT} \cdot |\{s_i \mid \exists m \in \text{UCB} : m \bmod c = s_i \wedge \exists m' \in \text{ECB} : m' \bmod c = s_i\}| \tag{3}$$

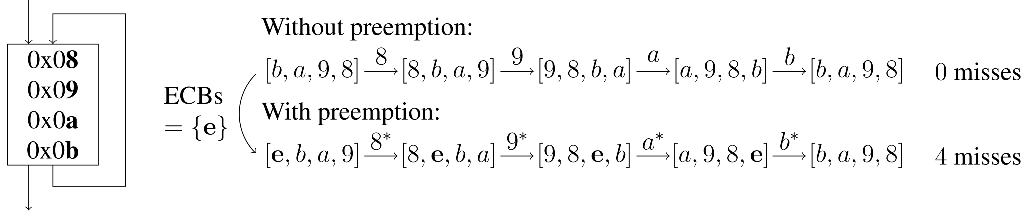


Figure 1. Evolution of the cache contents for LRU replacement. The first row shows the evolution of the cache contents accessing $8, 9, a, b$ without preemption. The second row shows the evolution of the cache contents on the same sequence with preemption. Preempting task accesses block e that maps to this cache set. Each miss is marked by *. Blocks 8, 9, a and b are useful before this access sequence.

The CRPD is bounded by the cache reload time times the number of sets, which at least one UCB and one ECB map to [3, 7].

The computation of UCBs and ECBs is out of the scope of this paper. Detailed information can be found in the original paper [2], as well as in [3, 6].

3. Bounding the CRPD for set-associative instruction caches

The definitions of UCBs and ECBs apply to all types of cache architectures, including set-associative caches with any replacement policy. Of course, whether a block is useful or not depends on the particular cache architecture, i.e., its associativity and replacement policy. So, a UCB analysis needs to be tailored to a specific cache architecture. In addition, for set-associative caches, the CRPD computation based on UCBs and ECBs differs from one replacement policy to another. As we show in this section, several pitfalls may be encountered on the way to the CRPD computation for the most common replacement policies LRU, FIFO, and PLRU.

3.1. LRU – Least-Recently-Used

Least-Recently-Used (LRU) policy replaces the least-recently-used element on a cache miss. It conceptually maintains a queue of length k for each cache set, where k is the associativity of the cache. In the case of LRU and associativity 4, $[b, c, e, d]$ denotes a cache set, where elements are ordered from most- to least-recently-used. If an element is accessed that is not yet in the cache (a miss), it is placed at the front of the queue. The last element of the queue, i.e., the least-recently-used, is then removed if the set is full. In our example, an access to f would thus result in $[f, b, c, e]$. Upon a cache hit, the accessed element is moved from its position in the queue to the front, in this respect treating hits and misses equally. Accessing c in $[f, b, c, e]$ results in $[c, f, b, e]$.

Since we concentrate on the use rather than on the computation of UCBs, we refer to Lee et al. [2] and Staschulat et al. [6] in case of LRU. For the derivation of the CRPD, there are again three possibilities: UCBs, ECBs and the combination of both.

The cache state after preemption may cause additional cache misses compared with the cache state without preemption. However the difference in the number of misses, and thus, the number of additional reloads, per cache set is always bounded by the associativity n: This is because any two initially different cache-set states converge within n accesses for LRU [4]. In particular, this holds for the cache states after preemption and the cache state without preemption. Furthermore, the number of additional misses is bounded by the number of reused memory blocks. Thus, the number of

UCBs bounds the additional cache misses.

However, due to control flow joins in the UCB analysis, the number of UCBs per set may exceed the associativity of the cache. Therefore, we have to limit the number of UCBs per set to n:

$$\text{CRPD}^{\text{LRU}}_{\text{UCB}} = \text{CRT} \cdot \sum_{s=1}^{c} min(|\text{UCB}(s)|, n) \tag{4}$$

where $\text{UCB}(s)$ denotes the set of UCBs mapping to cache set s.
By using solely the number of ECBs to bound the CRPD, a more complex formula is needed. As shown in Figure 1, a single ECB may cause up to n additional cache misses in the set it maps to. Therefore, the number of additional cache misses for set s ($\text{CRPD}^{\text{LRU}}_{\text{ECB}}(s)$) is zero, in case no ECB maps to set s, and, otherwise, bounded by the associativity (n) of the cache.

$$\text{CRPD}^{\text{LRU}}_{\text{ECB}} = \sum_{s=1}^{c} \text{CRPD}^{\text{LRU}}_{\text{ECB}}(s) \tag{5}$$

where

$$\text{CRPD}^{\text{LRU}}_{\text{ECB}}(s) = \begin{cases} 0 & \text{if ECB}(s) = \emptyset \\ \text{CRT} \cdot n & \text{otherwise} \end{cases} \tag{6}$$

where $\text{ECB}(s)$ denotes the set of ECBs mapping to cache set s.
To bound the number of additional reloads for cache set s using both UCBs and ECBs, Tan et al. [7] proposed the minimum function on the number of UCBs, the number of ECBs and the number of ways:

$$\text{CRPD}^{\text{LRU}}_{\text{MIN}}(s) = \text{CRT} \cdot \min(|\text{UCB}(s)|, |\text{ECB}(s)|, n) \tag{7}$$

where $\text{UCB}(s)$ and $\text{ECB}(s)$ denote the sets of UCBs and ECBs, respectively, mapping to cache set s.

However, this function gives an underestimation on the number of misses in several cases. Consider the CFG of Figure 1. All memory blocks map to the same cache set. Therefore, at the end of the execution of this basic block, the content of the 4-way set is given by $[b, a, 9, 8]$. As this basic block forms the body of a loop, these memory blocks are useful. One block of the preempting task maps to this set and evicts only one useful cache block: Using the minimum function, only one additional miss is taken into account for this memory set ($min(4, 1, 4) = 1$). However, the number of additional misses, four, is greater than the number of ECBs, one: All useful cache blocks are evicted and reloaded. In this example, the number of UCBs and the number of ways are upper bounds on the number of misses, but the minimum function results in an underestimation of the CRPD.

Instead of using the formula by Tan et al., the results from the CRPD computation via UCB and via ECB can be combined in a straight-forward manner:

$$\text{CRPD}^{\text{LRU}}_{\text{UCB\&ECB}} = \sum_{s=1}^{c} \text{CRPD}^{\text{LRU}}_{\text{UCB\&ECB}}(s) \tag{8}$$

where

$$\text{CRPD}^{\text{LRU}}_{\text{UCB\&ECB}}(s) = \begin{cases} 0 & \text{if ECB}(s) = \emptyset \\ \text{CRT} \cdot \min(|\text{UCB}(s)|, n) & \text{otherwise} \end{cases} \tag{9}$$

Again, $\text{UCB}(s)$ and $\text{ECB}(s)$ denote the sets of UCBs and ECBs, respectively, mapping to cache set s.

If no ECB maps to cache set s, no additional cache misses occur—as in Equation 6. Otherwise, the number of additional cache misses is bounded by the number of UCBs and the associativity of the cache—as in Equation 4.

3.2. FIFO – First-In, First-Out

First In, First Out (FIFO, also known as Round-Robin) bases its replacement decisions on *when* an element entered the cache, *not* on the time of its most-recent use. It replaces the element which has been resident in the cache for the longest time. FIFO cache sets can also be seen as a queue: new elements are inserted at the front evicting elements at the end of the queue. This resembles LRU. In contrast to LRU, hits do *not* change the queue. In our representation of FIFO cache sets, elements are ordered from last-in to first-in: Assume an access to f. In $[b, c, e, d]$, d will be replaced on a miss, resulting in $[f, b, c, e]$.

ECBs $= \{\mathbf{x}\}$

Without preemption: $$[b,a] \xrightarrow{a} [b,a] \xrightarrow{e^*} [e,b] \xrightarrow{b} [e,b] \xrightarrow{c^*} [c,e] \xrightarrow{e} [c,e] \quad \text{2 misses}$$

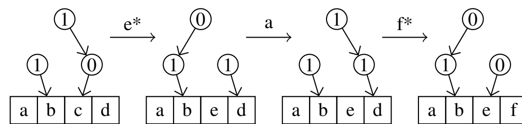
With preemption: $$[\mathbf{x},b] \xrightarrow{a^*} [a,\mathbf{x}] \xrightarrow{e^*} [e,a] \xrightarrow{b^*} [b,e] \xrightarrow{c^*} [c,b] \xrightarrow{e^*} [e,c] \quad \text{5 misses}$$

Figure 2. Evolution of the cache contents for FIFO replacement. The first row shows the evolution of the cache contents accessing a, e, b, c, e without preemption. The second row shows the evolution of the cache contents on the same sequence with preemption. Each miss is marked by *. Blocks a and b are useful before this access sequence.

The definitions of UCBs and ECBs also apply to FIFO caches. However, there is no correlation between the number of additional misses due to preemption and the number of UCBs, the number of ECBs or the number of ways. We illustrate this using the example of Figure 2. Consider a 2-way set-associative FIFO cache. The preemption occurs before the presented sequence: Blocks a and b are useful, memory block a is evicted and the final content of this set after preemption is $[x, b]$. The number of misses in the case without preemption is two and it is five in the case of preemption. The number of additional misses, three, is greater than the number of UCBs, two, the number of ways, two, and the number of ECBs, one. So, these numbers cannot be used as an upper bound on the number of additional misses when FIFO replacement is used.

3.3. PLRU – Pseudo-LRU

Pseudo-LRU (PLRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k - 1$ “tree bits” pointing to the line to be replaced next; a 0 indicating the left subtree, a 1 indicating the right. After every access, all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched. Consider the following example of 3 consecutive accesses to a set of a 4-way set-associative PLRU cache:



The first access in the sequence, to e, is a miss. It replaces c, which is pointed to by the tree bits. The tree bits on the path from e to the root of the tree are flipped to protect e from eviction. The second access in the sequence, to a, is a hit. The left tree bit already points away from a, so only the root tree

bit is flipped. Another access to a would not change the tree bits at all. Finally, the access to f, is another miss, replacing d.

As for FIFO replacement, the definition of UCB applies to the PLRU replacement strategy; but the number of UCBs, the number of ECBs and the number of ways are not upper bounds on the number of additional misses due to preemption. This is shown in Figure 3. There are two accesses to this set during the execution of the preempting task: Block b and d are evicted by x and y. The number of UCBs, four, the number of ECBs, two, and the number of ways, four, are lower than the number of additional misses, five. Thus, to compute an upper bound on the CRPD, one cannot simply use these numbers to derive an upper bound on the number of additional cache misses.

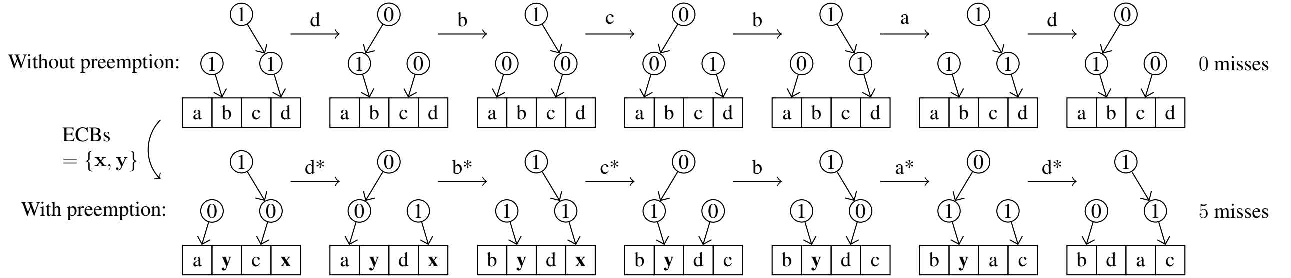


Figure 3. Evolution of the cache contents for PLRU replacement. The first row shows the evolution of the cache contents accessing d, b, c, b, a, d without preemption. The second row shows the evolution of the cache contents on the same sequence with preemption. Each miss is marked by *. Blocks a, b, c and d are useful before this access sequence.

For both FIFO and PLRU, we can extend the access sequences of our examples in Figures 2 and 3 such that the number of additional misses grows arbitrarily. In general, policies whose miss-sensitivity is greater than 1 exhibit such problems [4].

4. A new approach for FIFO, PLRU, and other policies

In the previous section, we have seen that the number of UCBs and the number of ECBs cannot be used to bound the CRPD for FIFO and PLRU. In this section, we *sketch* a new approach to the CRPD computation for policies other than LRU, including FIFO and PLRU. Due to space limitations, we cannot provide correctness proofs of the proposed approaches. Our basic idea is to transform guarantees for LRU to guarantees for other policies. To this end, we make use of the relative competitiveness of replacement policies:

4.1. Relative Competitiveness

Relative competitive analyses yield upper bounds on the number of misses of a policy P relative to the number of misses of another policy Q. For example, a competitive analysis may find out that policy P will incur at most 30% more misses than policy Q in the execution of any task.

For the definition of relative competitiveness we need the following notation: $m_P(p, s)$ denotes the number of misses of policy P on access sequence $s \in M^*$ starting in state $p \in C^P$. M denotes the set of memory blocks and C^P the set of states of policy P. For the precise definition of *compatible*, we refer the reader to [5]. Intuitively, it ensures that the policies are not given an undue advantage by the starting state.

Definition 3 (Relative Miss-Competitiveness)
A policy P is (k, c)-miss-competitive relative to policy Q, if

$$m_P(p, s) \leq k \cdot m_Q(q, s) + c$$

for all access sequences $s \in M^*$ and compatible cache-set states $p \in C^P, q \in C^Q$.

In other words, policy P will incur at most k times the number of misses of policy Q plus a constant c on any access sequence. For a pair of policies P and Q, there is a smallest k, such that P is k-miss-competitive relative to Q. This is called the *relative competitive ratio* of P to Q. Relative competitive ratios can be computed automatically for a pair of policies [5][2].

The competitiveness of FIFO and PLRU relative to LRU has been studied in [4, 5]. The most important results in our context are (the numbers in parentheses denote the associativities of the policies):

- PLRU(k) is $(1, 0)$-miss-competitive relative to LRU$(1 + log_2 k)$.
- FIFO(k) is $(\frac{k}{k-l+1}, l)$-miss-competitive relative to LRU(l).

This means that a PLRU-controlled cache of associativity k always performs at least as good as an LRU-controlled cache of associativity $1 + log_2 k$ with the same number of cache sets. In contrast to PLRU, it is not possible to achieve $(1, 0)$-miss-competitive relative to LRU for FIFO. However, the greater the associativity of FIFO is relative to the associativity of LRU, the closer $\frac{k}{k-l+1}$ gets to 1.

The relative competitiveness results mentioned above are for fully-associative caches. However, they can easily be lifted to set-associative caches, which can be seen as the composition of a number of fully-associative sets. If P is (k, c)-miss-competitive relative to Q in the fully-associative case, it is $(k, c \cdot s)$-miss-competitive relative to Q for a set-associative cache with s cache sets.

4.2. WCET and CRPD computation based on Relative Competitiveness

We have seen in the previous section that neither the number of UCBs nor the number of ECBs can be used to bound the CRPD for FIFO and PLRU. Our approach relies on the following observation: WCET and CRPD are always used in combination by schedulability analysis. For schedulability analysis it is sufficient, if (WCET bound + n·CRPD estimate) is greater than the execution time of the task including the cost of n preemptions. In that case, for safe schedulability analysis, the CRPD estimate alone does not have to bound the real CRPD. Recent work on CRPD computation provides only this weaker guarantee [1]. In addition, all sound existing approaches [2, 6] for LRU naturally also fulfill this weaker guarantee. For FIFO and PLRU, this allows us to compute a CRPD estimate that is not necessarily a bound on the CRPD itself: We will compute the WCET bound and CRPD estimate in such a way that together they provide the guarantee described above. The idea behind the approach we will present is to adapt WCET and CRPD analyses, which were originally designed for LRU, to other policies.

The CRPD accounts for all additional reloads due to preemption. Memory accesses that cause additional reloads due to preemption might have been considered to be hits by the WCET analysis, which assumes uninterrupted execution. The sequence of memory accesses along the execution of the preempted task can be split into two subsequences: the sequence of memory accesses before the

[2]See `http://rw4.cs.uni-sb.de/~reineke/relacs` for an applet that computes relative competitive ratios.

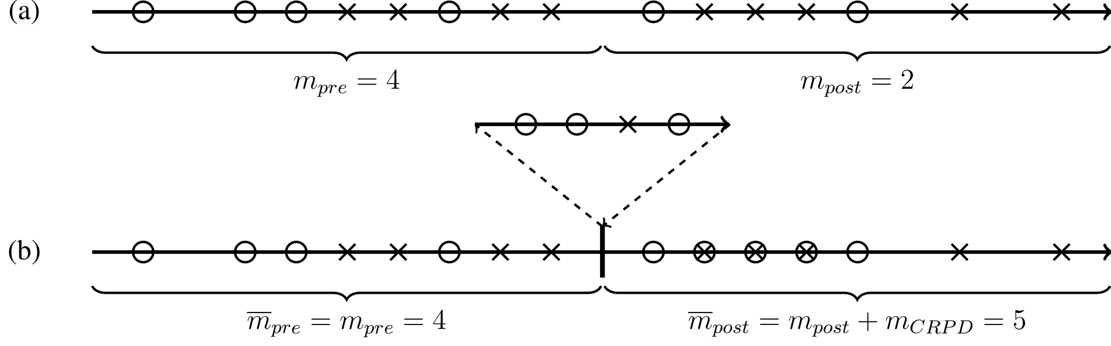


Figure 4. Sequence of memory accesses during task execution without preemption (a) and with preemption (b). $\bigcirc$ represents a miss, $\times$ a hit and $\otimes$ an additional miss due to preemption.

preemption point and the sequence after it. Figure 4 illustrates this situation. First we consider an uninterrupted task execution. Then, m_{pre} and m_{post} denote the number of misses on the sequences before and after the preemption point. For the interrupted execution, $\overline{m}_{pre}$ and $\overline{m}_{post}$ denote the number of misses on the same sequences including the effects of the preemption. As the execution of the first sequence is not influenced by the preemption, $\overline{m}_{pre}$ and m_{pre} are equal. m_{CRPD} is the number of additional misses due to preemption. So, $\overline{m}_{post}$ is the sum of m_{post} and m_{CRPD}. The total number of misses on the sequence is given by: $\overline{m} = \overline{m}_{pre} + \overline{m}_{post} = m_{pre} + (m_{post} + m_{CRPD}) = m + m_{CRPD}$.

For each of the two sequences, relative competitiveness allows to transfer the number of misses for one policy to a bound of a number of misses for another policy. Let policy $\mathrm{P}(t)$ be (k, c)-miss-competitive relative to $\mathrm{LRU}(s)$, and let $m^{\mathrm{P}(t)}$ and $m^{\mathrm{LRU}(s)}$ denote the number of misses of the two policies. Then,

$$\begin{aligned} \overline{m}_{pre}^{\mathrm{P}(t)} &\leq k \cdot \overline{m}_{pre}^{\mathrm{LRU}(s)} + c \\ &= k \cdot m_{pre}^{\mathrm{LRU}(s)} + c \end{aligned} \qquad \text{and} \qquad \begin{aligned} \overline{m}_{post}^{\mathrm{P}(t)} &\leq k \cdot \overline{m}_{post}^{\mathrm{LRU}(s)} + c \\ &= k \cdot (m_{post}^{\mathrm{LRU}(s)} + m_{CRPD}^{\mathrm{LRU}(s)}) + c. \end{aligned}$$

So, the total number of misses of policy $\mathrm{P}(m)$ can also be bounded by the number of misses of $\mathrm{LRU}(s)$:

$$\begin{aligned} \overline{m}^{\mathrm{P}(t)} &= \overline{m}_{pre}^{\mathrm{P}(t)} + \overline{m}_{post}^{\mathrm{P}(t)} \\ &\leq k \cdot m_{pre}^{\mathrm{LRU}(s)} + c + k \cdot (m_{post}^{\mathrm{LRU}(s)} + m_{CRPD}^{\mathrm{LRU}(s)}) + c \\ &= (k \cdot (m_{pre}^{\mathrm{LRU}(s)} + m_{post}^{\mathrm{LRU}(s)}) + c) + (k \cdot m_{CRPD}^{\mathrm{LRU}(s)} + c) \\ &= (k \cdot m^{\mathrm{LRU}(s)} + c) + (k \cdot m_{CRPD}^{\mathrm{LRU}(s)} + c). \end{aligned}$$

We can split this bound into $m^{\mathrm{P}(t)} = k \cdot m^{\mathrm{LRU}(s)} + c$ and $m_{CRPD}^{\mathrm{P}(t)} = k \cdot m_{CRPD}^{\mathrm{LRU}(s)} + c$, such that $\overline{m}^{\mathrm{P}(t)} \leq m^{\mathrm{P}(t)} + m_{CRPD}^{\mathrm{P}(t)}$. Note that while we *do* obtain a bound on the number of misses for $\mathrm{P}(t)$ including the preemption, namely $m^{\mathrm{P}(t)} + m_{CRPD}^{\mathrm{P}(t)}$, we do *not* necessarily obtain a bound on the cache-related preemption delay for $\mathrm{P}(t)$, i.e., $m_{CRPD}^{\mathrm{P}(t)}$ itself does not bound the real CRPD.

WCET and CRPD analyses take into account all possible access sequences. Our calculations above considered a single access sequence. However, the result can be lifted to all possible access sequences by considering bounds on the number of additional misses (CRPD) and bounds on the number of

misses during task execution. Also, we have considered exactly one preemption in our calculations. However, it can be verified rather easily, that $m^{\mathrm{P}(t)} + n \cdot m^{\mathrm{P}(t)}_{CRPD}$ is a bound on the number of misses for $\mathrm{P}(t)$ including up to n preemptions by similar calculations as above.

What does all this mean for the WCET and CRPD analysis of FIFO and PLRU? Due to its $(1,0)$-miss-competitive relative to $\mathrm{LRU}(1+log_2 k)$, $\mathrm{PLRU}(k)$ is particularly easy to treat: One can simply perform WCET and CRPD analyses assuming an $\mathrm{LRU}(1+log_2 k)$ cache. The resulting bounds will also be valid for $\mathrm{PLRU}(k)$. The situation becomes more difficult for FIFO. It is not $(1,0)$-miss-competitive relative to $\mathrm{LRU}(k)$ for any associativity. $\mathrm{FIFO}(k)$ is $(\frac{k}{k-l+1}, l)$-miss-competitive relative to $\mathrm{LRU}(l)$. To handle FIFO correctly, both the WCET and the CRPD analyses need to be adapted. $m^{\mathrm{FIFO}(k)}_{CRPD} = \frac{k}{k-l+1} \cdot m^{\mathrm{LRU}(l)}_{CRPD} + l$ can be used as a CRPD estimate for $\mathrm{FIFO}(k)$ given that $m^{\mathrm{LRU}(l)}_{CRPD}$ is a sound estimate for $\mathrm{LRU}(l)$. Likewise, the WCET analysis for $\mathrm{FIFO}(k)$ needs to account for $\frac{k}{k-l+1}$ as many misses as it would have to for $\mathrm{LRU}(l)$ plus l misses. For WCET analyses that explicitly compute an upper bound on the number of cache misses, like ILP-based approaches, this can be rather easily. Other WCET analyses that take cache misses into account as part of the execution time of basic blocks it maybe more difficult to account for the higher number of misses.

5. Conclusions and Future Work

Prior useful cache block analyses mainly focus on direct-mapped caches. They use the number of UCBs and/or ECBs to derive an upper-bound on the CRPD. As we have shown in this paper, considering set-associative caches, the CRPD computation is much more complex. In case of LRU replacement, the computation of the CRPD solely based on the number of UCBs is correct, whereas a previously proposed formula combining UCBs and ECBs may underapproximate the CRPD. In contrast to LRU, for PLRU and FIFO policies, neither UCBs nor ECBs can be used to bound the CRPD.

For LRU, we introduce a new CRPD formula based on UCBs and ECBs. For other policies, we sketch a new approach to bound the CRPD. This approach is based on the concept of relative competitiveness: Bounds obtained for LRU are transformed into bounds for other policies. For future work we plan to implement and experimentally evaluate this approach.

References

[1] ALTMEYER, S., AND BURGUIÈRE, C. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)* (July 2009), IEEE Computer Society, pp. 109–118.

[2] LEE, C.-G., HAHN, J., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *RTSS'96* (1996), IEEE Computer Society, p. 264.

[3] NEGI, H. S., MITRA, T., AND ROYCHOUDHURY, A. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03* (2003), ACM.

[4] REINEKE, J. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, , Saarbrücken, November 2008.

[5] REINEKE, J., AND GRUND, D. Relative competitive analysis of cache replacement policies. In *LCTES'08* (June 2008), ACM, pp. 51–60.

[6] STASCHULAT, J., AND ERNST, R. Scalable precision cache analysis for real-time software. *ACM Trans. on Embedded Computing Sys. 6*, 4 (2007), 25.

[7] TAN, Y., AND MOONEY, V. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES'04* (2004), pp. 182–199.

[8] TOMIYAMA, H., AND DUTT, N. D. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES'00* (2000), ACM.

COMPARISON OF IMPLICIT PATH ENUMERATION AND MODEL CHECKING BASED WCET ANALYSIS

Benedikt Huber and Martin Schoeberl[1]

Abstract
In this paper, we present our new worst-case execution time (WCET) analysis tool for Java processors, supporting both implicit path enumeration (IPET) and model checking based execution time estimation. Even though model checking is significantly more expensive than IPET, it simplifies accurate modeling of pipelines and caches. Experimental results using the UPPAAL *model checker indicate that model checking is fast enough for typical tasks in embedded applications, though large loop bounds may lead to long analysis times. To obtain a tool which is able to cope with larger applications, we recommend to use model checking for more important code fragments, and combine it with the IPET approach.*

1. Introduction

Worst-case execution time (WCET) analysis is needed as input to schedulability analysis. As measuring the WCET is not a safe approach, real-time tasks have to be analyzed statically. We present a WCET analysis tool for Java. The first target processor that is supported by the tool is the Java processor JOP [24]. A future version will also support the multi-threaded Java processor jamuth [27].

The WCET analysis is performed on Java bytecodes, the intermediate representation of compiled Java programs. A Java virtual machine (JVM) executes bytecodes by either interpreting the bytecodes, compiling them to native code, or executing them in hardware. The latter case is the execution model of a Java processor. Using bytecode as the instruction set without further transformation simplifies WCET analysis, as the execution time can be modeled at the bytecode level.

The primary method for estimating the WCET is based on the standard IPET approach [21]. We have implemented a context sensitive, bottom-up analysis, and an interprocedural analysis to support a simple method cache approximation. The tool additionally targets the UPPAAL model checker [3]. The control flow graphs (CFG) are modeled as timed automata, and a binary search is used to obtain the WCET. An (exact) cache simulation was added to the UPPAAL model, which is used to evaluate the quality of static cache approximations. Having implemented those two methods in a single tool, we compare traditional IPET with model checking based techniques.

Although we analyze Java programs, neither the model checking based analysis nor the method cache are Java specific (e.g., the method cache has been implemented in the CarCore processor [15]).

[1]Institute of Computer Engineering, Vienna University of Technology, Austria;
email: benedikt.huber@student.tuwien.ac.at, mschoebe@mail.tuwien.ac.at

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2281

1.1. The Target Architecture JOP

JOP [24] is an implementation of the Java virtual machine (JVM) in hardware. The design is optimized for low and predictable WCETs. Advanced features, such as branch prediction or out-of-order execution, that are hard to model for the WCET analysis, are completely avoided. Instead, the pipeline and the caches are designed to deliver good real-time performance.

Caches are a mandatory feature for pipelined architectures to feed the pipeline with enough instructions and speedup load/store instructions. Although direct-mapped caches or set-associative caches with a least recently used (LRU) replacement policy are WCET friendly, the instruction cache in JOP has a different organization. The cache stores full methods and is therefore named method cache [23]. The major benefit of that cache is that cache misses can only occur at method invoke or on a return from a method. All other instructions are guaranteed hits and the cache can be ignored during WCET analysis.

JOP uses a *variable block method cache*, where methods are allowed to occupy more than one cache block. JOP currently implements the variable block cache using a so called next-block replacement, effectively a first in, first out (FIFO) strategy. It is known, that the FIFO replacement strategy is not ideal for WCET analysis [22]. A LRU replacement would be preferable, as it provides a better caching behavior and is easier to analyze. However, the constraint on the method cache that a method has to span several contiguous blocks, makes the implementation of a LRU replacement strategy difficult.

1.2. Related Work

WCET related research started with the introduction of timing schemas by Shaw [26]. Shaw presents rules to collapse the CFG of a program until a final single value represents the WCET. An overview of actual WCET research can be found in [20, 28]. Computing the WCET with an integer linear programming solver is proposed in [21] and [13]. The approach is named graph-based and implicit path enumeration respectively. We base our WCET analyzer on the ideas from these two groups.

Cache memories for the instructions and data are classic examples of the paradigm: *Make the common case fast*. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Plenty of effort has gone into research to integrate the instruction cache into the timing analysis of tasks [1, 9] and cache analysis integration with the pipeline analysis [8]. Heckmann et. al described the influence of different cache architectures on WCET analysis [10]. Our approach to cache analysis is to simplify the cache with the method cache. That form of caching needs no integration with the pipeline analysis and the hit/miss categorization can be approximated at the call graph, which leads to shorter analysis times.

WCET analysis at the bytecode level was first considered in [4]. It is argued that the well formed intermediate representation of a program, the Java bytecode, is well suited for a portable WCET analysis tool. In that paper, annotations for Java and Ada are presented to guide the WCET analysis at bytecode level. The work is extended in [2] to address the machine-dependent low-level timing analysis. Worst-case execution frequencies of Java bytecodes are introduced for a machine independent timing information. Pipeline effects (on the target machine) across bytecode boundaries are modeled by a *gain time* for bytecode pairs. Due to our target architecture that executes Java bytecode natively we can extend on the work of WCET analysis at bytecode level.

In [25], an IPET based WCET analysis tool is presented that includes the timing model of JOP. A simplified version of the method cache, the two block cache, is analyzed for invocations in inner loops. Trevor Harmon developed a tree-based WCET analyzer for interactive back-annotation of WCET estimates into the program source [6]. The tool is extended to integrate JOP's method cache [7] in a similar way as in [25]. Compared to those two tools, which also target JOP, our presented WCET tool is enhanced with: (a) analysis of bytecodes that are implemented in Java; (b) a tighter method cache analysis; and (c) an evaluation of model checking for WCET analysis and exact modeling of the method cache in the model checking approach.

Whether and to what extend model checking can deliver tighter WCET bounds than IPET is investigated in [16]. Though we use timed automata and a different cache model, the argument that state exploration is potentially more precise than IPET still applies. A project closely related to our model checking approach is presented in [5]. Model checking of timed automata is used to verify the schedulability of a complete application. However, even with a simple example, consisting of two periodic and two sporadic tasks, this approach leads to a very long analysis time.

2. The WCET Analysis Tool

The new WCET analysis tool deals with a less restricted subset of Java than [25], adding support for bytecodes implemented in Java and dynamic dispatch. The supported subset of the Java language is restricted to acyclic callgraphs. As any static WCET analysis, we require the set of classes to be known at compile time.

For determining loop bounds, we rely on both source code annotations (similar to the ones described in [11, 25]) and dataflow analysis [19]. The latter is also used for computing the set of methods possibly executed when a virtual method is invoked. By default, the WCET is calculated using IPET, taking the variable block method cache into account (see Section 2.1.). Additionally, a translation to timed automata has been implemented, including method cache simulations (see Section 2.2.).

2.1. IPET and Static Cache Approximation

The primary method for estimating the WCET is based on the standard IPET approach [21]. Hereby, the WCET is computed by solving a maximum cost circulation problem in a directed graph representing the program's control flow. Each edge is associated with a certain cost for executing it, and linear constraints are used to exclude infeasible paths.

In addition to the usual approach modeling the application as one big integer linear program (ILP), we also support a bottom-up analysis, which first analyses referenced methods separately and then solves the intraprocedural ILP model. This is a fast, simple alternative to the global model, and allows us to invoke the model checker for smaller, important parts of the application. Furthermore, the recursive analysis is able to support incremental updates, and could be used by a bytecode optimizer or an interactive environment.

After completing the WCET analysis, we create detailed reports to provide feedback to the user and annotate the source code as far as possible. For this purpose, the solution of the ILP is analyzed, first annotating the nodes of the CFG with execution costs, and then mapping the costs back to the source code.

Computing the Cache Cost To compute the cost of cache misses for JOP, we observe that a cache miss either occurs when executing an invoke instruction or when control returns to the caller. Each method takes a fixed number of cycles to load (depending on its size), and depending on the instruction triggering the cache miss, a certain number of those load cycles are hidden.

FIFO caches show unbounded memory effects [14] and simulating the cache with an initially empty cache is not a safe approximation. This applies even when the cache sequence corresponds to a path in a non-recursive call tree, so it is necessary to *flush the cache* at the task's entry point to get a safe approximation using simulation. As furthermore a long access history is needed to classify a cache access as hit or as miss, standard dataflow analysis techniques perform poorly on FIFO caches [22].

Approximating the Method Cache using Static Analysis Our technique to approximate the cache miss cost for an N-block FIFO method cache is based on the following fact: If it is known that during the execution of some method m, at most N distinct cache blocks (including those of m) are accessed, each accessed block will be loaded *at most once* when executing m. Currently a simple heuristic is used to identify such *all-fit* methods, by checking whether all methods possibly invoked during the execution of m fit into the cache.

To include this approximation into the ILP model, we traverse the callgraph of the program starting from the root method, and check for each call site, whether the invoked method m is *all-fit*. If this is the case, the supergraph of m is duplicated, adding constraints that the code of each method possibly accessed during the execution is loaded at most once. Otherwise, the invoke and return cache access are considered to be a cache miss.

2.2. Calculating the WCET using Model Checking

UPPAAL is a model checker based on networks of timed automata, supporting *bounded integer variables* and synchronization using *channels* [3]. We have implemented a translation of (Java) programs to UPPAAL models, and use the model checker to determine a safe WCET bound. This allows us to potentially deal with complex timing dependencies, and gives us the opportunity to verify whether and to what extent model checking works for practical WCET estimation. Our initial attempt was based on ideas from [17, 5] and has been subsequently refined using progress measures and adding cache simulations.

General Strategy An UPPAAL model comprises global declarations of clocks, synchronization channels and variables, and a set of *processes*. Each process is instantiated from a *template* and may have its own set of local clocks and variables. We start by declaring a global clock t, which represents the total time passed so far, and build timed automata simulating the behavior of the program, one for each method. Additionally, we add a clock representing the local time passed at some instruction, t_{local}.

There is one location I, which is the entry point of the program, and one location E, which corresponds to the program's exit. When execution has finished, the system takes the transition from E to the final state EE (Figure 1). If we want to check whether t_{guess} is a safe WCET bound, we ask the model checker to verify that for all states which are at location E, $t \leq t_{guess}$ holds. If this property is satisfied, t_{guess} is a safe WCET bound, otherwise we have to assume it is not. Starting with a known

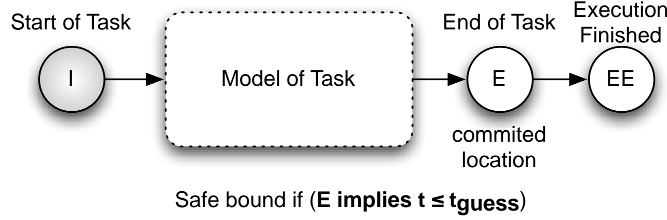
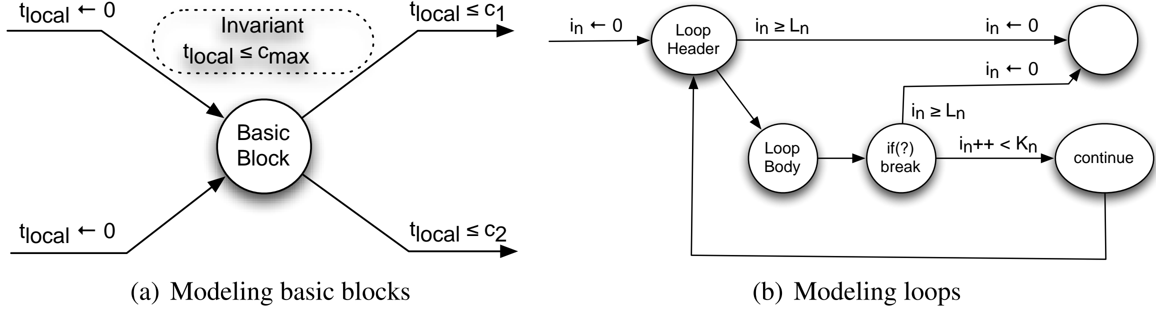


Figure 1. Calculating the WCET bound using UPPAAL



(a) Modeling basic blocks (b) Modeling loops

Figure 2. Modeling CFGs as Timed Automata

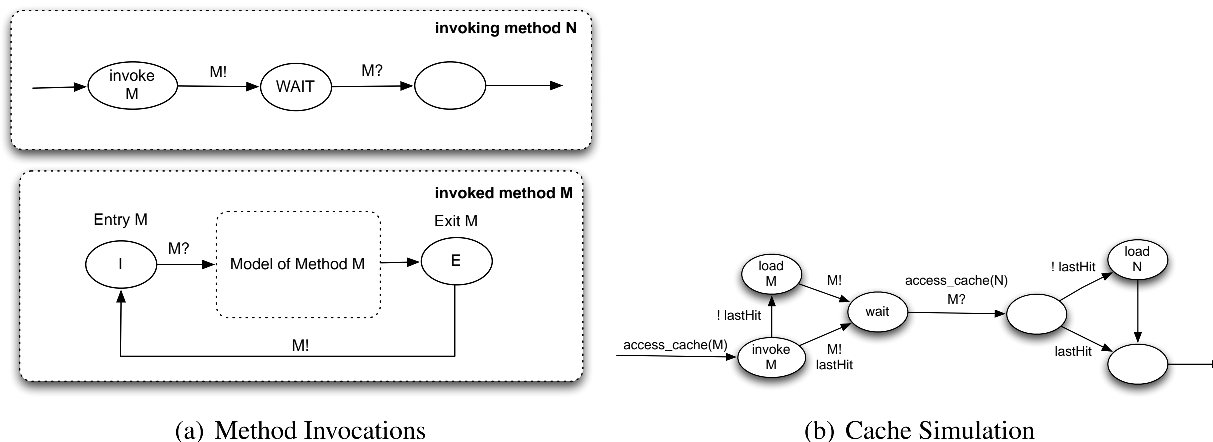
upper bound, provided by the IPET analysis, we perform a binary search to find a tighter WCET bound. Note that if the model checker kept track of the maximum value of t encountered during state exploration, it would not be necessary to perform a binary search.

Translation of CFGs Given the CFG of a Java method m_i, we build an automaton M_i simulating the behavior of that method by adding *locations* representing the CFG's nodes and *transitions* representing the flow of control. The initial location $M_i.I$ corresponds to the entry node of the CFG, and the location $M_i.E$ to its exit node.

To model the timing of basic blocks, we reset t_{local} at the incoming edges of a basic block. If the execution of the basic block takes at most c_{max} cycles, we add the invariant $t_{local} \leq c_{max}$ to the corresponding location. On architectures where the maximum number of cycles depends on the edge e taken, additional guards of the form $t_{local} \leq c_e$ are added to the outgoing edges of the basic block (Figure 2(a)).

Modeling Loops It would be possible to eliminate bounded loops by unrolling them in a preprocessing step, but it is more efficient to rely on bounded integer variables. Assume it is known that the body of loop n is executed at least L_n and at most K_n times. We declare a local bounded integer variable i_n representing the loop counter, ranging from 0 to K_n. The loop counter is initialized to 0 when the loop is entered. If an edge implies that the loop header will be executed one more time, a guard $i_n < K_n$ and an update $i_n \leftarrow i_n + 1$ is added to the corresponding transition. If an edge leaves the loop, we add a guard $i_n \geq L_n$ and an update $i_n \leftarrow 0$ to the transition (Figure 2(b)).

It might be beneficial to set $L_n = K_n$, but this is only correct in the absence of timing anomalies, and therefore in general unsound in the presence of FIFO caches. In principle, every control flow repre-



(a) Method Invocations (b) Cache Simulation

Figure 3. Translating Method Invocations and Cache Accesses

sentable using bounded integer variables can be modeled using UPPAAL, though we only implemented simple loop bounds in the current version of our tool.

Method Invocations We build one automaton M_i for each reachable method m_i. To model method invocations, we synchronize the method's automata using channels. When a method m_i is invoked, the invoke transition synchronizes with the outgoing transition of $M_i.I$ on the invoked method's channel. When returning from method m_i, the transition from $M_i.E$ to $M_i.I$ synchronizes with the corresponding return transition in the calling method. This translation assumes that there are no recursive calls. To allow the method to be invoked several times, a transition from $M_i.E$ to $M_i.I$ is added to all methods (see Figure 3(a)).

Method Cache Simulation Using timed automata, it is possible to directly include the cache state into the timing model. It is most important, however, to keep the number of different cache states low, to limit space and time needed for the WCET calculation. JOP's method cache is especially well suited for model checking, as the number of blocks and consequently the number of different cache states are small.

To include the method cache in the UPPAAL model, we introduce an array of global, bounded integer variables, representing the blocks of the cache. It is assumed that the cache initially only contains the main method and is otherwise empty. As this is not a safe approximation in general, we have to ensure that the first access to some method is actually a cache miss, for example by inserting a cache flush at the beginning of the main method.

We insert two additional locations right before and after the invoke location, modeling the time spent for loading the invoked method and the invoking method at a return, respectively. The UPPAAL function `access_cache` updates the global cache state and sets the variable `lastHit` to either true or false (see Figure 3(b) and Listing 1).

```
1  const int NBLOCKS[NMETHODS] = { /* number of blocks per method */ };
2  const int UNDEF = NMETHODS;
3  /* In the initial state , the main method occupies the first , say, K blocks of
4     the cache. Therefore, all fields of the array are undefined, except the one
5     at position K−1, which is set to the id of the main method. */
6  int [0,NMETHODS] cache[NBLOCKS] = { UNDEF, ..., MAIN_ID, UNDEF, ... };
7  bool lastHit ;                        /* whether last access was cache hit */
8  void access_cache (int mid) {
9      int p = 0;                        /* pointer into the cache */
10     int mblocks = NBLOCKS[mid]; /* blocks of accessed method */
11     lastHit = false ;                 /* no cache hit so far */
12
13     /* Check if mid is in the cache */
14     for(p = 0; p < NBLOCKS; p++)
15         if (cache[p] == mid) { lastHit = true ; return; }
16
17     /* Move cache blocks and insert new tag */
18     for(p = NBLOCKS − 1; p >= mblocks; p−−)
19         cache[p] = cache[p − mblocks];
20     for(p = 0; p < sz−1; p++)
21         cache[p] = UNDEF;
22     cache[p] = mid;                   /* tag is written at position mblocks − 1 */
23 }
```

Listing 1. FIFO variable block cache simulation

Progress Measures and Optimizations The performance of the model checker crucially depends on the search order used for state exploration. Instead of using the default breadth-first search, performance is greatly improved by using an *optimal search order*. For a loop-free CFG, it is obviously beneficial to deal with node *a* before node *b*, if *a* is executed before *b* on each possible execution path. This topological order is generalized to CFGs with loops by taking the values and bounds of loop counters into account. Finally, we extend this idea to applications with more than one method by defining a relative *progress measure* [12], which is incremented on transitions and *monotonically increases* on each possible execution path. The progress measure both guides UPPAAL's state space exploration and reduces memory usage.

Additionally, for cache simulation purposes we pre-calculate the WCET of inner loops and leaf methods, using either model checking or IPET. The corresponding subgraphs of the CFG are replaced by summary nodes, resulting in a locally simplified model. Currently, we are developing a model checker prototype to explore further optimizations using state abstractions, optimized data structures, and exploiting sharing, caching, and locality.

3. Evaluation

In this section, we will present some experimental results obtained with our tool. The first problem set[2] consists of small benchmarks to evaluate the model checker's performance for intraprocedural analysis. The second set comprises benchmarks extracted from real world, embedded applications. Those examples are reactive systems, with many control flow decisions but few loops, making it difficult to obtain a good execution time estimate using measurements.

Table 1 lists the number of methods, the bytecode size of the tasks under consideration, and the number of control flow nodes in the (conceptually) unrolled supergraph. Additionally, we list the size of the largest method in bytes, which determines the minimal cache size. As WCET analysis targets

[2]Provided by the Mälardalen Real-Time Research Center, except GCD.

Problem	Description	Methods	Size (Total / Max)	Nodes
DCT	Discrete Cosine Transform (8×8)	2	968 / 956	45
GCD	Greatest Common Divisor (32 bit)	3	138 / 100	9599
MatrixMult	Matrix Multiplication (50×50)	3	106 / 84	19460
CRC	Cyclic Redundancy Check (40 bytes)	6	404 / 252	26861
BubbleSort	Bubble Sort (500×500)	2	87 / 80	1000009
LineFollower	A simple line-following robot	9	226 / 76	96
Lift	Lift controller	13	1206 / 216	438
UdpIp	Network benchmark	28	1703 / 304	9600
Kfl	*Kippfahrleitung* application	46	2539 / 1052	11348

Table 1. Problem sets for evaluation

Problem	Measured ET JOP	Single Block IPET	FIFO Cache IPET	FIFO Cache UPPAAL	Pessimism Ratio
DCT	19124	19131	19131	19124	1.00
GCD	62963	75258	73674	73656	1.17
MatrixMult	1.09M	1.09M	1.09M	1.09M	1.00
CRC	0.19M	0.47M	0.38M	0.38M	2.00
BubbleSort	32.16M	46.33M	46.33M	46.33M	1.44
LineFollower	2348	2610	2411	2368	1.03
Lift	5484	8897	8595	8355	1.57
UdpIp	8375	131341	130518	129638	15.58
Kfl (8 Block)	10616	49744	40452	37963	3.81

Table 2. Measured and computed WCET

single tasks, the size of the considered applications seems to be realistic for embedded systems. On the other hand, we would definitely benefit from a larger and varying set of benchmarks.

3.1. Comparison of Measured Execution Time and Calculated WCET

Table 2 compares the measured execution times and the computed WCET estimates. For all experiments, we use a memory access timing of 2 cycles for a read and 3 cycles for a write. We use a 16-block variable-block FIFO method cache, with 1 KB instruction cache memory in total.

The WCET was computed assuming a cache in which every access is a cache miss (Single Block IPET), using the static method cache approximation described in Section 2.1. (FIFO Cache IPET), and the UPPAAL cache simulation presented in the last section (FIFO Cache UPPAAL). [3]

Pessimistic estimations in the first problem set are mainly due to missing flow facts and data dependent flow (`BubbleSort`, `CyclicRedundancyCheck`), while in the second problem set the measurements do not cover all execution paths (`Lift`, `Kfl`) or use a small payload (`UdpIp`). The estimates using the static cache approximation are quite close to the exact cache simulation using UPPAAL, so here, the approximation worked well. One should note, however, that the cache costs are in general small, and a bigger difference could occur on other platforms.

[3] For the `Kfl` benchmark, UPPAAL ran out of memory, so we had to reduce the number of cache blocks to 8.

Problem	IPET	UPPAAL Breadth First		UPPAAL Progress	
		Verify	Search	Verify	Search
DCT	0.00	0.09	1.21	0.07	0.84
GCD	0.00	3.10	47.65	0.20	2.75
MatrixMult	0.01	0.21	3.52	0.23	4.58
CRC	0.01	1.21	16.32	0.52	9.46
BubbleSort	0.00	7.63	197.91	10.33	268.43
LineFollower	0.00	0.11	1.07	0.15	1.10
Lift	0.01	0.17	2.07	0.18	1.38
UdpIp	0.03	8.98	84.69	1.78	30.28
UdpIp simplified		0.64	7.28	0.44	7.07
Kfl (no cache)	0.04	92.19	1229.42	0.57	9.23
Kfl simplified (no cache)		33.81	444.73	0.45	7.23
Kfl (8 blocks)	0.13	—	—	31.77	428.24
Kfl simplified (8 blocks)		—	—	17.72	263.18

Table 3. Analysis execution time in seconds

3.2. Performance Analysis

We have evaluated the time needed to estimate the WCET using the techniques presented in this paper, summarized in Table 3. For the approach using the UPPAAL model checker, we consider the translation without (UPPAAL Breadth First) and with progress measures (UPPAAL Progress Measure). For the two largest problems, we additionally consider locally simplified models (Section 2.2.). The time spent in the binary search and the maximum time needed for one invocation of the UPPAAL verifier are listed in the table. For comparison, we measure the time spent in the ILP solver, when using the global IPET approach.

All experiments were carried out using an Intel Core Duo 1.83 Ghz with 2 GB RAM on `darwin-9.5`. For IPET, the linear programs were solved using `lp_solve` 5.5. We use UPPAAL 4.0.7 with aggressive space compression (`-S 2`), but without convex hull approximation, as for our translation, the number of generated states is not affected by this choice.

For the first problem set, the analysis times for the IPET approach are below 10 milliseconds, as loops can be encoded efficiently using linear constraints. We observe that UPPAAL handles the analysis of those small benchmarks well too, as long as the number of executions of the innermost loop's body do not get too large (as in `BubbleSort`).

In the second problem set, the solver times for the IPET approach are still below 0.2 seconds, so the additional equations for the cache approximation did not result in a significant increase of the time needed for the ILP solver. Although model simplifications reduced the analysis times for the larger two problems, the most important optimization for UPPAAL is the use of progress measures. Progress measures can greatly improve performance (up to -99%), sometimes leading to slightly longer analysis times ($+35\%$). Furthermore, the cache simulation of the `Kfl` benchmark was only possible using progress measures, although we had to reduce the number of cache blocks to avoid a state explosion.

4. Conclusion

In this paper, we presented a WCET analysis tool for Java processors. The tool is open-source under the GNU GPL license.[4] The tool includes a static analysis of the effects of the method cache. The first target is the Java processor JOP. In a future version we will include jamuth as a second target.

From the comparison of IPET with model checking based WCET analysis we see that IPET analysis outperforms model checking analysis with respect to the analysis time. However, model checking allows easy integration of complex hardware models, such as that of the method cache. We conclude that a combined approach where model checking is used on simplified problems delivers tight WCET bounds at a reasonable analysis time. In the current version of the tool we simplified CFGs and used model checking for the global analysis.

As future work we consider using model checking to analyze local effects and solve the global analysis with IPET. As an application of this approach we consider model checking for the complex timing interaction of a chip-multiprocessor version of JOP [18].

Acknowledgement

One author thanks Anders P. Ravn for suggesting UPPAAL for the model checking based WCET analysis during an enjoyable research visit at the University of Aalborg. We thank Anders for the initial UPPAAL model of the method cache and constructive comments on preliminary versions of the paper. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design).

References

[1] ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium* (1994), pp. 172–181.

[2] BATE, I., BERNAT, G., MURPHY, G., AND PUSCHNER, P. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications* (Dec. 2000), pp. 39–48.

[3] BEHRMANN, G., DAVID, A., AND LARSEN, K. G. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004* (September 2004), M. Bernardo and F. Corradini, Eds., no. 3185 in LNCS, Springer–Verlag, pp. 200–236.

[4] BERNAT, G., BURNS, A., AND WELLINGS, A. Portable worst-case execution time analysis using Java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems* (Jun 2000).

[5] BOGHOLM, T., KRAGH-HANSEN, H., OLSEN, P., THOMSEN, B., AND LARSEN, K. G. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)* (New York, NY, USA, 2008), ACM, pp. 106–114.

[4] The source is available on the project's website `http://www.jopdesign.com`.

[6] HARMON, T., AND KLEFSTAD, R. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)* (August 2007).

[7] HARMON, T., SCHOEBERL, M., KIRNER, R., AND KLEFSTAD, R. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)* (St. Louis, MO, United States, April 2008).

[8] HEALY, C. A., ARNOLD, R. D., MUELLER, F., WHALLEY, D. B., AND HARMON, M. G. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers 48*, 1 (1999), 53–70.

[9] HEALY, C. A., WHALLEY, D. B., AND HARMON, M. G. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium* (1995), pp. 288–297.

[10] HECKMANN, R., LANGENBACH, M., THESING, S., AND WILHELM, R. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE 91*, 7 (Jul. 2003), 1038–1054.

[11] HU, E. Y.-S., KWON, J., AND WELLINGS, A. J. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003* (February 2003), vol. LNCS 2968, pp. 208–228.

[12] KRISTENSEN, L. M., AND MAILUND, T. A generalised sweep-line method for safety properties. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right* (London, UK, 2002), Springer-Verlag, pp. 549–567.

[13] LI, Y.-T. S., AND MALIK, S. 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* (New York, NY, USA, 1995), ACM Press, pp. 88–98.

[14] LUNDQVIST, T. *A WCET analysis method for pipelined microprocessors with cache memories*. PhD thesis, Chalmers University of Technology, Sweden, 2002.

[15] METZLAFF, S., UHRIG, S., MISCHE, J., AND UNGERER, T. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)* (New York, NY, USA, 2008), ACM, pp. 38–45.

[16] METZNER, A. Why model checking can improve WCET analysis. In *Computer Aided Verification (CAV)* (Berlin/Heidelberg, 2004), vol. 3114 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 334–347.

[17] OUIMET, M., AND LUNDQVIST, K. Verifying execution time using the TASM toolset and UPPAAL. Tech. Rep. Embedded Systems Laboratory Technical Report ESL-TIK-00212, Embedded Systems Laboratory Massachusetts Institute of Technology.

[18] PITTER, C. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[19] PUFFITSCH, W. Supporting WCET analysis with data-flow analysis of Java bytecode. Research Report 16/2009, Institute of Computer Engineering, Vienna University of Technology, Austria, February 2009.

[20] PUSCHNER, P., AND BURNS, A. A review of worst-case execution-time analysis (editorial). *Real-Time Systems 18*, 2/3 (2000), 115–128.

[21] PUSCHNER, P., AND SCHEDL, A. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems 13*, 1 (Jul. 1997), 67–91.

[22] REINEKE, J., GRUND, D., BERG, C., AND WILHELM, R. Timing predictability of cache replacement policies. *Journal of Real-Time Systems 37*, 2 (Nov. 2007), 99–122.

[23] SCHOEBERL, M. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)* (Agia Napa, Cyprus, October 2004), vol. 3292 of *LNCS*, Springer, pp. 371–382.

[24] SCHOEBERL, M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2* (2008), 265–286.

[25] SCHOEBERL, M., AND PEDERSEN, R. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)* (New York, NY, USA, 2006), ACM Press, pp. 202–211.

[26] SHAW, A. C. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng. 15*, 7 (1989), 875–889.

[27] UHRIG, S., AND WIESE, J. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)* (New York, NY, USA, 2007), ACM Press, pp. 230–237.

[28] 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. *Trans. on Embedded Computing Sys. 7*, 3 (2008), 1–53.

EXTENDING THE PATH ANALYSIS TECHNIQUE TO OBTAIN A SOFT WCET

Paul Keim, Amanda Noyes, Andrew Ferguson
Joshua Neal, Christopher Healy[1]

Abstract

This paper discusses an efficient approach to statically compute a WCET that is "soft" rather than "hard". The goal of most timing analysis is to determine a guaranteed WCET; however this execution time may be far above the actual distribution of observed execution times. A WCET estimate that bounds the execution time 99% of the time may be more useful for a designer in a soft real-time environment. This paper discusses an approach to measure the execution time distribution by a hardware simulator, and a path-based timing analysis approach to derive a static estimation of this same distribution. The technique can find a soft WCET for loops having any number of paths.

1. Introduction

In order to properly schedule tasks in a real-time or embedded system, an accurate estimate of each task's worst-case execution time (WCET) is required. To fulfill this need, a static timing analyzer is used to estimate the WCET. This WCET is guaranteed to be greater than or equal to the actual execution time no matter what the input data may be. However, in some cases this predicted WCET may be quite loose, because the probability of taking the worst-case path all the time may be minuscule.

There is a distinction between a hard and a soft real-time system. In a hard real-time system, deadlines are absolute, and thus the WCET estimate must never be less than the actual execution time. Timing analysis is typically concerned with this case of finding this "hard WCET." By contrast, in a soft-real time system, such as media production, an occasional deadline may be tolerated. This paper is concerned with the notion of a "soft WCET" – a WCET estimate that is "almost" always above the actual execution time, one that is guaranteed to be correct at least, say, 99% of the time.

It has been customary for WCET analysis to address hard real-time constraints. However, it could be the case that the WCET is much higher than the actual execution times experiences. Even the 99th percentile of the execution time distribution could be much lower than the computed WCET. And if this is the case, the WCET bound would be overly conservative in a soft real-time environment, in which an occasional missed deadline could be tolerated.

[1] Department of Computer Science, Furman University, Greenville SC 29613 USA; e-mail: chris.healy@furman.edu

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2292

The main contribution of this research is to determine an efficient means to compute a soft WCET, and we do so by first determining the whole distribution of execution times. Probability distributions are continuous functions, so to make our task manageable, we quantize this distribution by creating an array of buckets, in which each bucket represents one possible execution time. In practice, we selected an array size of 100 buckets, so that we in effect store percentiles of the execution time distribution. The last bucket, containing the highest value, will contain the "soft WCET". Analogously, the first bucket would contain the lowest value or "soft BCET", but our study was focused only on WCET.

2. Related work

This paper is inspired by the work of other researchers to apply statistics and probability techniques to timing analysis and real-time systems. Specifically, the pioneering work of Bernat et al. in 2002 proposed a stochastic or probabilistic approach to WCET analysis [2]. Stochastic analysis means we want to look at the overall distribution of execution times. Then, we can observe where the WCET lies in relation to the bulk of this distribution. Another statistical approach was proposed by Edgar [4], in which he uses the Gumbel distribution to infer the WCET from a sample of data points given that the distribution of execution times is unknown in advance. Atlas and Bestavros even used similar statistical techniques in order to generalize and relax the classical Rate Monotonic Scheduling algorithm [1]. This paper differs from other recent approaches mainly in that we are concerned with creating a data structure that can capture the overall distribution of execution times, rather than focusing on the extreme tail of the distribution.

3. A Simple Example

Figure 1 shows a simple scenario. Suppose we want to compare the WCET with the overall execution time distribution for the following simple loop:

```
for (i = 0; i < 100; ++i)
  if (a[i] > 0)
    ++count;
```

After compiling, the `if`-statement becomes a conditional branch that can either be taken or not. Thus, this loop has two paths, A and B. The probability of taking each path is p_A and p_B respectively. We also need to know *n*, the number of loop iterations, which is 100 in this example. We can now easily compute the probability of taking path A *k* times out of *n* iterations using the binomial probability formula:

$$prob = \frac{n!}{k!(n-k)!} \; p_A^k p_B^{n-k}$$

If we know nothing about the values in the array a, we assume there is an equal chance of taking either path. Static timing analysis can tell us that the execution times of these two paths are 6 and 12 cycles. Therefore the execution time for the loop is between 600 and 1200, and we obtain the results in Figure 1. Had the probabilities been unequal, the overall shapes would be almost

unchanged: they would merely be shifted to the left or right and skewed (i.e. asymmetric) to a small degree. The extreme execution times at the tails would still practically never be attained.

Note that if a loop has only one path, it would not have an interesting distribution. Both the probability and execution time distribution graphs would have a single spike with a height of 1.00. If a loop has more than two paths, then there is the potential for multiple peaks in the distributions.

The point to take away from this example is that the WCET, which is about 1200 cycles, is very unlikely to be achieved. There is more than a 99% chance that the actual execution time will be less than 1000 cycles. No amount of loop analysis can tighten the hard WCET estimate without further knowledge of the distribution of values of the array. On the other hand, a soft WCET calculation can bound the execution time distribution much more tightly.

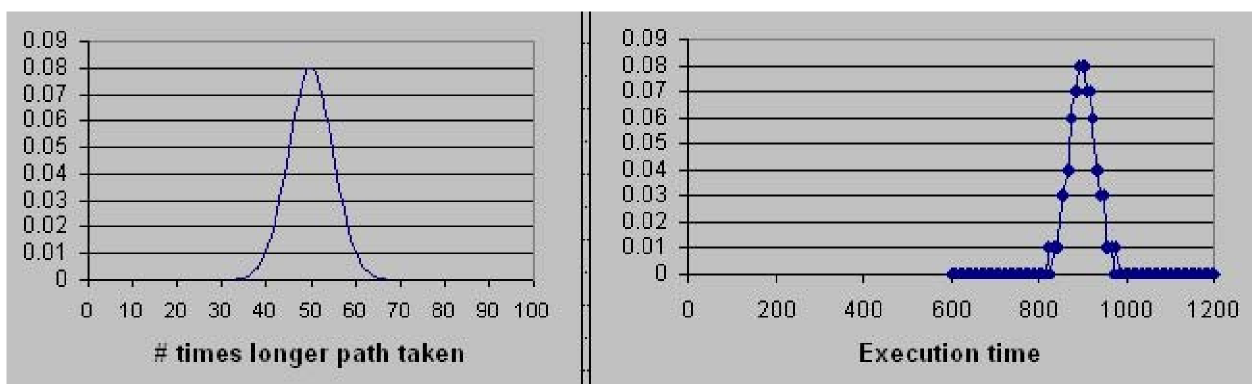


Figure 1: Probability and execution time distributions.

4. Overview of the Approach

Our approach utilizes two separate tools: a hardware simulator and timing analyzer. Details on the background of our approach can be found elsewhere [5]. The purpose of the simulator is to check the timing analyzer's predictions for accuracy. We use a simulator so that it is straightforward to make changes to the hardware specification, and it is straightforward to pinpoint sub-execution times for portions of the program, if necessary. We adapted the simulator to run a benchmark executable with randomized input values. After a set of trials, it then creates a histogram of the execution time distribution. The right tail of this distribution is the observed WCET of the trials.

Our methodology for running the simulator was straightforward: we would run each benchmark 1000 times with randomized input data. To make things more interesting we considered "fair" versus "unfair" distributions of input data. By "fair" we mean that the input data was generated in such a way that the paths would logically be taken about an equal number of times. For example, if there is a single if-statement checking to see if an array element is positive, then we would arrange a "fair" test to ensure that the array elements had a 50% chance of being positive. To generate "unfair" distributions, we made one path 3 times as likely to be executed.

The second tool is the timing analyzer that we used is an adaptation of an earlier version that computes hard WCET's. It relies on control-flow information determined from a compiler. Then, it represents the input program in a tree that hierarchically organizes the constituent functions, loops, basic blocks and instructions. Each loop's path information is then created and also stored with the

appropriate loop. The evaluation of the WCET proceeds in a bottom-up fashion, starting at the lowest level of detail (the instruction) and eventually working its way to the `main()` function. One important part of the timing analysis approach is the loop analysis which must examine the execution paths.

In working with both the hardware simulator (which measures execution time dynamically) and the timing analyzer (which estimates execution time statically), our goal is that the soft WCET predicted by the timing analyzer should be close to the longest execution time observed by the simulation.

5. Loop Analysis Algorithm

How do we statically generate a distribution of execution times? To illustrate our approach, let us assume that we have a loop with *n* iterations. If the loop has only one execution path, then this case is not interesting, since the execution time will not depend on the input data. In this trivial case, the execution time distribution would be completely flat and uniform. There will only be a meaningful distribution of execution times if the loop has two or more paths.

A loop's execution time array is created. In practice we chose our execution time array to hold 100 values, but this is simply an input parameter to the algorithm and can easily be changed. In our case, each cell represents 1 percentile of the distribution of execution times for the loop in question. We have found that this is sufficiently granular to capture a nearly continuous distribution.

The following algorithm illustrates the case of a loop having exactly two paths.

Objective: to estimate the execution time distribtion of a loop
Let n be the number of iterations.
First, find execution time of each path, which is assumed to be a constant number of cycles.
Second, assign probabilities to each path. Infeasible paths are given a probability of 0.0 and not included. Otherwise, we assume each path is equally likely to be taken.
Of the 2 paths, let's call A the longer path, and B the shorter.

```
previous_cumulative_prob = 0
For i = 0 to n:
        prob = binomial probability that path A executes i times and B executes n-i times.
        cumulative_prob += prob
        for each integer j between 100*previous_cumulative_prob and 100*cumulative_prob
                time_dist[ j ] = time taking A i times + B (n-i) times.
```

If a loop has three or more paths, there are two possible approaches. One approach is to repeatedly apply our binomial probabilities on each conditional branch we encounter. Otherwise, we can attempt to assign every path a separate probability and apply a multinomial probability distribution. Formulas involving multinomial probabilities can become quite complex [9], so we decided on the former approach.

As a simple illustration, here is a motivation for how we would handle 3 paths using the binomial probability method. Our three paths are called p1, p2, and p3. Paths p1 and p2 have a 25% each of being chosen, while p3 has a 50% chance of being chosen. What we can do is take the distribution

graph of p1 and p2, generated through binomial distribution of an equal probability using the 2-path algorithm above (both are 25%, but since they're the only ones we're allowing to run, it's actually 50%, a coin flip). Then, with the distribution {p1, p2}, we combine this with {p3}. Both now have a 50% probability, so we take the 100 values of each array, merge them into an array having 200 values, sort them in ascending order, and then *remove every other element* so that we end up with 100 values again, and this becomes our distribution array for all three paths.

Here is the algorithm for the general case of 3 or more paths in a loop:

Let $\oplus$ represent the application of the earlier binomial probability formula. We also need to weight the distribution if the probabilities of the paths are different in the following manner:

> **a. Weight the number of elements. Suppose the probability associated with each path is such that P2>P1. Thus, the 2nd path's array continues to contain 100 elements, and the combined array will stretch to include (100*P2/P1) elements.**
>
> **b. To create the extra elements for the combined distribution, starting with the initial 100 elements, duplicate accordingly. If needing 200 elements, duplicate each element. If needing 150 elements, duplicate every other element.**

Let $\{P_1,P_2,\ldots P_n\}$ be the set of paths, each with the probability of being executed P. We create the probability distribution D as follows.

1. **Use the binomial probability on $\{P_1,P_2\}$ to initialize D.**
2. **For i = 3 to n, $D = D \oplus P_i$**

This elegant solution scales to any number of paths, because it is a repeated application of the binomial probability formula, rather than using a much more complex multinomial one.

6. Results

In our methodology, we assume that the loop under consideration reads from up to two globally declared arrays. These arrays may be multi-dimensional, and may contain either integers or floating-point values. We created 1000 versions of the benchmark, each with a different set of values for initializing the array(s). The initialization is done at the global declaration so that it does not impact the execution time of the benchmark under review. The benchmarks we examined were the same as those used in previous work.

One could note that there is a potential for some lack of precision. There is always a chance that running a program 1000 times will miss the absolute worst case input data. However, our results show that in most cases, the distribution of execution times is relatively tight. This can happen for two reasons. First, the conditional-control flow that depends on the program's input data can be drowned out by the rest of the program that does not depend on the input data. Second, it is often the case that when there are two paths, their execution times do not differ a great deal.

Actually, we found that testing our timing analyzer against the simulator was a little easier to automate than in our previous work with "hard" WCET's. Before, we would compare the static prediction of the WCET with a single measurement. That measurement used a specific input which we wrote by hand, studying it to make sure it is the worst-case input data. However, in general this is not always feasible. This is another reason why we decided to adopt a stochastic approach.

There may be benchmarks for which it may be very difficult to determine the input data that will drive every worst-case path throughout the program. But with a stochastic approach, you can observe the probability distribution and focus your attention on the highest bucket (e.g. the 99th percentile).

Table 1 shows some of the benchmarks we ran, and how they are characterized based on their observed stochastic behavior. These benchmarks have been used in previous work [6]. The numbers in parentheses indicate the execution time of the longest trial of that benchmark, expressed as a percentage of its statically predicted absolute WCET. If this value is much below 100, this does not indicate that the static WCET prediction is inaccurate. Rather, it indicates that the actual WCET, i.e. taking every opportunity to execute a worst-case path, is very rarely encountered (and indeed, was not encountered in the 1000 trials). Note that in the case of the very narrow distributions, the skewness of the distribution is insignificant.

What is interesting about these results is that the same benchmark with different input data (fair versus unfair) can have a markedly different distribution of execution times. For example, in the case of the Sym benchmark, which tests a matrix to see if it is symmetric, a fair distribution yields a simulated execution time that was never more than 2% of the hard WCET. But our unfair distribution that made one path 3 times as likely to be taken brought up the maximum observed execution time to 19% of the hard WCET.

	Width 0-2%	Width 3-25%	Width >= 26%
Skewed +	Sumnegallpos-fair (98) Summatrix-fair (93) Sym-fair (2)	Sumnegallpos-unfair (100) Sym-unfair (19)	Sprsin-unfair (98)
Skewed -	Neville (80)	Summatrix-unfair (100)	Bubblesort-fair (85)
No skew	Unweight-fair (95)	Sprsin-fair (74) Unweight-unfair (100)	Bubblesort-unfair (86) Sumoddeven (79)

Table 1: Stochastic characteristics of selected benchmark programs.

Additionally, the number of paths affects the distribution. We were primarily interested in benchmarks with 2 paths. Generally this means we could expect one peak in the distribution, and we could use the theory of binomial probability distribution to predict their behavior. However, it is interesting to note that even benchmarks with many execution paths still had just one observable peak in the distribution. Other peaks could exist, but they may be relatively insignficant.

The analysis of a single program can become more interesting when you combine the effect of more than one loop. For example, the `Sprsin` benchmark has three loops, each with a different execution time distribution classification.

- Loop 1's distribution is uniformly around 100%.
- Loop 2's distribution is tightly between 77 and 83%.
- Loop 3's distribution varies widely between 67 and 90%.

The overall behavior of sprsin is similar to loop 2.

To illustrate the robustness of the loop analysis algorithm of Section 5 to a non-trivial number of paths, we show an example benchmark containing a loop with 7 paths. Its source code is given below. Figure 2 shows the two execution time distributions: the 1000 trials from the simulator, compared against the histogram of the 100 buckets created by the timing analyzer.

```
int a[60][60], pos, neg, zero;

main() {
  int i;

  for (i = 0; i < 60; ++i) {
    if (a[i][0] > 0 && a[0][i] > 0)
      ++pos;
    else if (a[i][0] == 0 && a[0][i] == 0)
      ++zero;
    else
      ++neg;
  }
}
```

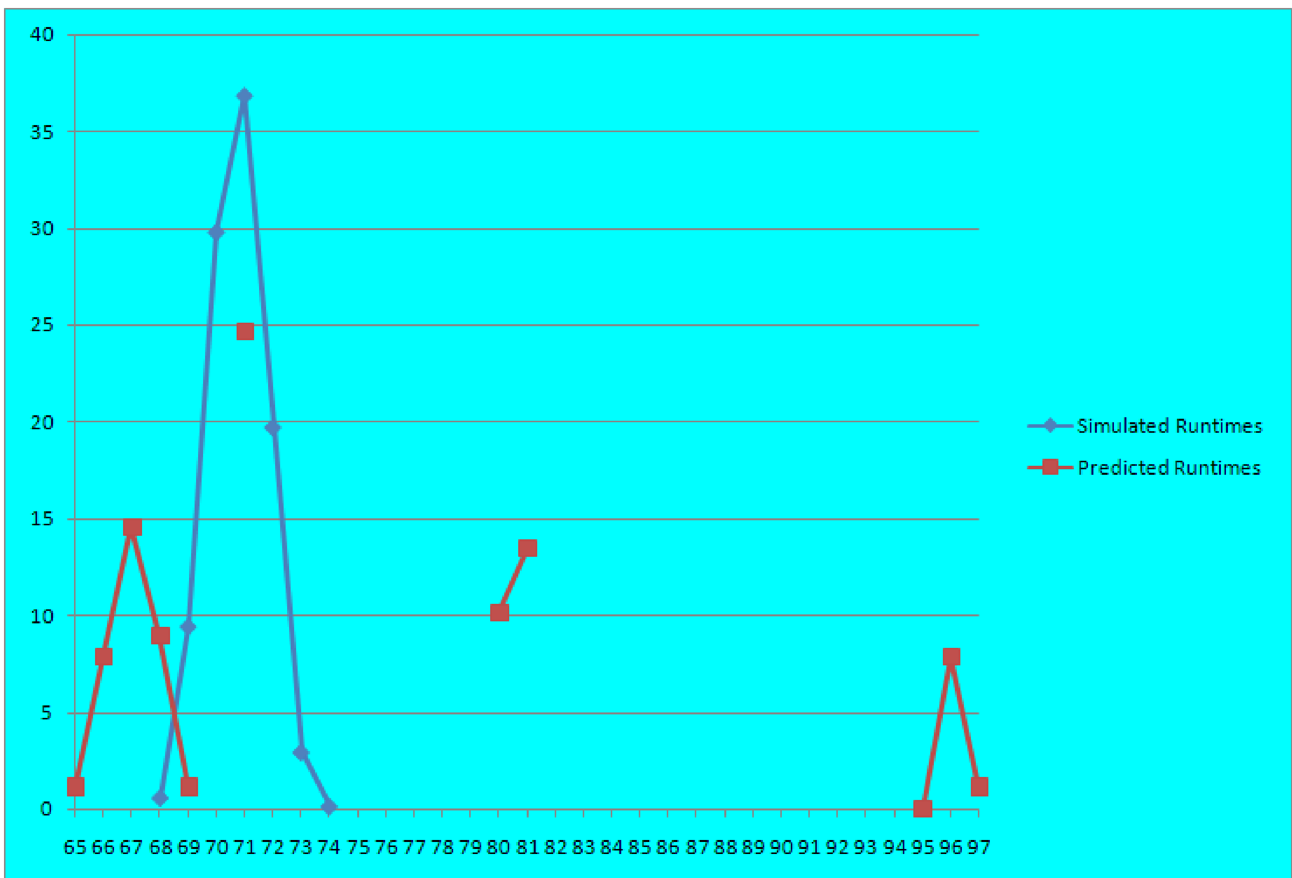


Figure 2: Percentage histogram of the simulated and predicted runtimes for the 7-path benchmark example. The horizontal axis refers to the percentage of the hard WCET, and the vertical axis is the percentage of trials.

The observed execution times never exceeded 74% of the hard WCET. Meanwhile the soft WCET predicted by the timing analyzer was 97% of the hard WCET. This example illustrates a current limitation to our approach: we assume each conditional branch taken 50% of time, which is not always realistic. This accounts for much of the discrepancy. The timing analyzer can do a better job if it has some knowledge of the probability of a certain branch being taken.

7. Future Work

Currently, our approach works for single loops containing any number of paths. So, naturally, the next step would be to handle multiple loops in a program, whether nested or consecutive. For example, we could make use of the convolution technique described by Bernat et al. [2] to compose multiple execution time distributions. In order to make the timing analyzer's execution time distribution (and its soft WCET) more accurate, we also need to specifically analyze branch instruction. For example, loop branches are usually taken, and we can at least use a better heuristic in such a case.

Another direction of future work is a refinement of how we handle probability. Currently, we treat iterations independently, and the choice of path on each iteration is random. In the future, we can take „runs" into account [8], *i.e.* the fact that the same path may be taken many times in succession. In particular, if the same (long) path is taken for many consecutive iterations, this will temporarily bias the execution time distribution upward, closer to the absolute WCET. For example, in a loop with 1000 iterations, it may be interesting to examine the distribution of execution times of any set of 50 consecutive iterations of the loop.

In addition, we can also combine our stochastic approach with our recent work on parametric timing analysis [3]. Currently, our approach here assumes that the execution times are constant values and that the number of loop iterations is known at compile time.

8. Conclusion

A timing analyzer's static analysis of a program's absolute, or hard, WCET can often be far in excess of the execution times actually realized at run time. Thus, in the context of soft real-time deadlines, it may be worthwhile to alternatively consider a WCET that is not absolute, but at the 99th percentile, or other sufficiently high cutoff value. This value we call the soft WCET. The authors have modified an existing timing analyzer and simulator to determine the execution time distribution between the BCET and WCET. The work in modifying the simulator opened our eyes to the fact that these distributions are quite varied in their width, skewness and maximum value.

However, the major contribution of this paper is the update to the static timing analyzer, which uses the traditional path analysis technique. Rather than computing just a single BCET or WCET value, it can determine execution times at each percentile of probability. The highest value in this distribution is the soft WCET estimate. The approach scales to allow any number of paths. In practice, working with our benchmark suite, the maximum number of paths encountered in the loops was 17. The utility of this work is that the timing analyzer can quickly report both an absolute WCET and a soft WCET. If there is a large difference between these two values, and the application resides in a soft real-time system, then a variety of measures can be undertaken to exploit this gap. For example, dynamic voltage scaling techniques can be used to reduce the clock speed and thus significantly reduce the energy consumption [7]. In a heterogeneous multicore system, the application can be assigned to a slower processor. Finally, if the application is performing some iterative computation, more iterations can be safely performed.

9. Acknowledgements

The authors are grateful for the contributions of two of our colleagues. Colby Watkins implemented the graphical user interface that integrates the output from both the timing analyzer and hardware simulator. William Graeber ported the GUI driver to PHP, making it possible analyze code from a remote site without having to install the timing tools. The authors also thank Peter Puschner and Iain Bate for their helpful suggestions. This research was supported in part by the Furman Advantage Research Program.

10. References

[1] ATLAS, A. K. and BESTAVROS, A. „Statistical Rate Monotonic Scheduling," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1998, pp. 123-132.

[2] BERNAT, G., COLIN, A., PETTERS, S., „WCET Analysis of Probabilistic Hard Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, December 2002, pp. 279-288.

[3] COFFMAN, J., HEALY, C., MUELLER, F., WHALLEY, D, „Generalizing Parametric Timing Analysis," *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems*, June 2007, pp. 152-154.

[4] EDGAR, S. F. *Estimation of Worst-Case Execution Time Using Statistical Analysis*, PhD Thesis, University of York, 2002.

[5] HEALY, C., ARNOLD, R., MUELLER, F., WHALLEY, D., HARMON, M., „Bounding Pipeline and Instruction Cache Performance," *IEEE Transactions on Computers*, January 1999, pp. 53-70.

[6] HEALY, C. and WHALLEY, D., „Automatic Detection and Exploitation of Branch Constraints for Timing Analysis," *IEEE Transactions on Software Engineering*, August 2002, pp. 763-781.

[7] MOHAN, S., MUELLER, F., HAWKINS, W., ROOT, M., HEALY, C., WHALLEY, D., „Parascale: Exploiting Parametric Timing Analysis for Real-Time Schedulers and Dynamic Voltage Scaling," *Proceedings of the IEEE Real-Time Systems Symposium*, December 2005, pp. 233-242.

[8] ROSS, S., *A First Course in Probability*, Macmillian, New York, 1988.

[9] WISHART, J. „Cumulants of Multivariate Multinomial Distributions," *Biometrica* **36** (1/2) June 1949, pp. 47-58.

FROM TRUSTED ANNOTATIONS TO VERIFIED KNOWLEDGE [1]

Adrian Prantl[2], Jens Knoop[2], Raimund Kirner[3], Albrecht Kadlec[3] and Markus Schordan[4]

Abstract
WCET analyzers commonly rely on user-provided annotations such as loop bounds, recursion depths, region- and program constants. This reliance on user-provided annotations has an important drawback. It introduces a Trusted Annotation Basis into WCET analysis without any guarantee that the user-provided annotations are safe, let alone sharp. Hence, safety and accuracy of a WCET analysis cannot be formally established. In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds. Fundamental to our approach is to apply model checking in concert with other more inexpensive program analysis techniques, and the coordinated application of two algorithms for Binary Tightening *and* Binary Widening*, which control the application of the model checker and hence the computational costs of the approach. Though in this paper we focus on the control of model checking by Binary Tightening and Widening, this is embedded into a more general approach in which we apply an array of analysis methods of increasing power and computational complexity for proving or disproving relevant time bounds of a program. First practical experiences using the sample programs of the Mälardalen benchmark suite demonstrate the usefulness of the overall approach. In fact, for most of these benchmarks we were able to empty the trusted annotation base completely, and to tighten the computed WCET considerably.*

1. Motivation

The computation of loop bounds, recursion depths, region- and program constants is undecidable. It is thus commonly accepted that WCET analyzers rely to some extent on user-assistance for providing bounds and constants. Obviously, this is tedious, complex, and error-prone. State-of-the-art approaches to WCET analysis thus provide for a fully automatic preprocess for computing required bounds and constants using static program analysis. This unburdens the user since his assistance is reduced to bounds and constants, which cannot automatically be computed by the methods employed by the preprocess. Typically, these are classical data-flow analyses for constant propagation and folding, range analysis and the like, which are particularly cost-efficient but may fail to verify a bound or the constancy of a variable or term. WCET analyzers then rely on user-assistance to provide the missing bounds which are required for completing the WCET analysis. This introduces a *Trusted Annotation Base* (*TAB*) into the process of WCET analysis. The correctness (safety) and optimality

[2]Vienna University of Technology, Institute of Computer Languages, Austria, {adrian,knoop}@complang.tuwien.ac.at
[3]Vienna University of Technology, Institute of Computer Engineering, Austria, {raimund,albrecht}@vmars.tuwien.ac.at
[4]University of Applied Sciences Technikum Wien, Austria, schordan@technikum-wien.at
[1]This work has been 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 and within the research project "Sustaining Entire Code-Coverage on Code Optimization" (SECCO) under contract No P20944-N13, and by the European Union within the 7th EU R&D Framework Programme within the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068.

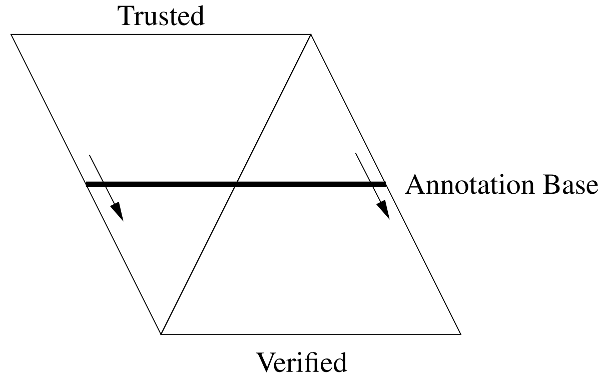


Figure 1. The annotation base: shrinking the trusted annotation base and establishing verified knowledge about the program

(tightness) of the WCET analysis depends then on the safety and tightness of the bounds of the TAB provided by the user.

In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds.

Figure 1 illustrates the general principle of our approach. At the beginning the entire annotation base that is added by the user where static analysis fails to establish the required information, is assumed and trusted to be correct, thus we call it *Trusted Annotation Base* (*TAB*). Using model checking we aim to verify as many of these user-provided facts as possible. In this process we shrink the trusted fraction of the annotation base and establish a growing verified annotation base. In Figure 1 the current state in this process is visualized as the horizontal bar. In our approach we are lowering this bar, representing the decreasing fraction of trust to an increasing fraction of verified knowledge, and thus transfer *trusted user-belief* into *verified knowledge*.

2. Shrinking the Trusted Annotation Base – Sharpening the Time Bounds

2.1. Shrinking the Trusted Annotation Base

The automatic computation of bounds by the preprocesses of up-to-date approaches to WCET analysis is a step towards keeping the trusted annotation base small. In our approach we go a step further by shrinking the trusted annotation base. In practice, we often succeed to empty it completely.

A key observation is that a user-provided bound – which the preprocessing analyses were unable to compute – can not be checked by them either. Hence, verifying the correctness of the corresponding user annotation in order to move it *a posteriori* from the trusted annotation base to the verified knowledge base requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for the detection of copy constants, linear constants, simple constants, conditional constants, up to finite constants detecting different classes of constants at different costs [9]. This provides evidence for the variety of available choices for analyses using the example of constant propagation and folding. While some of these algorithms might in fact well be able to verify a user annotation, none of these algorithms is especially prepared and suited for solely verifying a data-flow fact at a particularly chosen program location, a so-called *data-flow query*. This is because these algorithms are exhaustive in nature. They are designed to analyze whole programs.

They are not focused towards deciding a data-flow query, which is the domain of *demand-driven program analyses* [3, 6]. Like for the more expressive variants of constant propagation and folding, however, demand-driven variants of program analyses are often not available.

In our approach, we thus propose to use *model checking* for the *a posteriori* verification of user-provided annotations. Model checking is tailored for the verification of data-flow queries. Moreover, the development of software model checkers made tremendous progress in the past few years and are now available off-the-shelf. The Blast [1] and the CBMC [2] model checkers are two prominent examples. In our experiments reported in Section 4 we used the CBMC model checker. The way we use model checking is similar to the approach of Rieder et al. who also used model checking to derive loop bounds [13]. However, their basic motivation of using model checking is to minimize the implementation effort. Since they use model checking as the only program analysis technique, they conclude that model checking in general is a very costly analysis technique. This builds an excellent motivation for our approach where we use model checking in concert with other program analysis techniques of different complexity.

The following example demonstrates the ease and elegance of using a model checker to verify a loop bound, which we assume could not be automatically bounded by the program analyses used. The program fragment on the left-hand side of Figure 2 shows a loop together with a user-provided annotation of the loop. The program on the right-hand side shows the transformed program which is presented to CBMC to verify or refute the user-provided annotation:

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */


  while(low <= up){
#pragma wcet_trusted_loopbound(7)
    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;


  }


  return fvalue;
}
```

$\longrightarrow$

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */

  unsigned int __bound = 0;
  while(low <= up){

    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;

    __bound += 1;
  }
  assert(__bound <= 7);

  return fvalue;
}
```

Figure 2. Providing loop bound annotations for the model checker

In this example the CBMC model checker comes up with the answer “yes,” i.e., the loop bound provided by the user is safe; allowing thus for its movement from the trusted to the verified annotation base. If, however, the user were to provide $__bound \leq 3$ as annotation, model checking would fail and produce a counter example as output. Though negative, this result would still be most valuable. It allows for preventing usage of an unsafe trusted annotation base in a subsequent WCET analysis. Note that the counter example itself, which in many applications is the indispensable and desired output of a failed run of a model checker, is not essential for our application. It might be useful, however, to present it to the user when asking for another candidate of a bound, which can then be subject to *a posteriori* verification in the same fashion until a safe bound is eventually found.

Next we introduce a more effective approach to come up with a safe and even tight bound, if so existing, which does not even rely on any user interaction. Fundamental for this are the two algorithms *Binary Tightening* and *Binary Widening* and their coordinated interaction. The point of this coordination is to make sure that model checking is applied with care as it is computationally expensive.

2.2. Sharpening the Time Bounds

2.2.1. Binary Tightening

Suppose a loop bound has been proven safe, e.g. by verifying a user-provided bound by model checking or by a program analysis. Typically, this bound will not be tight. In particular, this will hold for user-provided bounds. In order to exclude channeling an unsafe bound into the trusted annotation base, the user will generously err on the side of caution when providing a bound. This suggests the following iterative approach to tighten the bound, which is an application of the classical pattern of a binary search algorithm, thus called *binary tightening* in our scenario.

Let b_0 denote the value of the initial bound, which is assumed to be safe. Per definition b_0 is a positive integer. Then: call procedure $binaryTightening$ with the interval $[0..b_0]$ as argument, where $binaryTightening([low..high])$ is defined as follows:

1. Let $m = \lceil \frac{low+high}{2} \rceil$.
2. ModelCheck(m is a safe bound):
3. yes: $low = m$: **return** m
   $low = m - 1$: ModelCheck(low is a safe bound)
   yes: **return** low no: **return** m
   otherwise: $binaryTightening([low..m])$
4. no: $high = m$: **return** $false$
   $high = m + 1$: ModelCheck($high$ is a safe bound)
   yes: **return** $high$ no: **return** $false$
   otherwise: $binaryTightening([m..high])$

Obviously, $binaryTightening$ terminates. If it returns $false$, a safe bound tighter than that of the initial bound b_0 could not be established. Otherwise, i.e., if it returns value b, this value is the least safe bound. This means b is tight. If it is smaller than b_0, we succeeded to sharpen the bound.

Binary widening described next allows for proceeding in the case where a safe bound is not known *a priori*. If a safe bound (of reasonable size) exists, binary widening will find one, without any further user interaction.

2.2.2. Binary Widening

Binary widening is dual to binary tightening. Its functioning is inspired by the risk-aware gambler playing roulette, who exclusively bets on 50% chances like red and black. Following this strategy, in principle, any loss can be flattened by doubling the bet the next game. In reality, the maximum bet allowed by the casino or the limited monetary resources of the gambler, whatever is lower, prevent this

strategy to work out in reality. Nonetheless, the idea of an externally given limit yields the inspiration for the *Binary Widening* algorithm to avoid looping if no safe bound exists. A simple approach is to limit the number of recursive calls of binary widening to a predefined maximum number. The version of binary widening we present below uses a different approach. It comes up with a safe bound, if one exists, and terminates, if the size of the bound is too big to be reasonable, or does not exist at all. This directly corresponds to the limit set by a casino to a maximum bet.

Let $b_0 \geq 1$ be a positive integer, and let max be the maximum value for a safe bound considered reasonable. Then: Call procedure $binaryWidening$ with b_0 and max as arguments, where $binaryWidening(b, limit)$ is defined as follows:

1. if $b > limit$: **return** false
2. ModelCheck(b is a safe bound):
3. yes: **return** b
4. no: $binaryWidening(2 * b, limit)$

Obviously, $binaryWidening$ terminates.[1] If it returns $false$, at most a bound of a size considered unreasonably big exists, if at all. Otherwise, i.e., if it returns value b, this value is a safe bound. The ratio behind this approach is the following: if a safe bound exists, but exceeds a predefined threshold, it can be considered practically useless. In fact, this scenario might indicate a programming error and should thus be reported to the programmer for inspection. A more refined approach might set this threshold more sophisticatedly, by using application dependent information, e.g., such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time this loop shall be allowed for.

2.2.3. Coordinating Binary Tightening and Widening

Once a safe bound has been determined using binary widening, binary tightening can be used to compute the uniquely determined safe tight bound. Because of the exponential resp. logarithmic behaviour in the selection of arguments for binary widening and tightening, model checking is called moderately often. This, together with the application of other program analyses, which leaves model-checking to the "hard" cases, is the key for the practicality of our approach. We implemented this approach in our WCET analyzer TuBound, as described in Section 3. The results of practical experiments we conducted with the prototype implementation are promising. They are reported in Section 4.

3. Implementation within TuBound

3.1. TuBound

TuBound [11] is a research WCET analyzer tool working on a subset of the C++ language. It is unique for uniformly combining static program analysis, optimizing compilation and WCET calculation. Static analysis and program optimization are performed on the abstract syntax tree of the input

[1] In practice, the model checker might run out of memory before verifying a bound, if it is too large, or may take too much time for completing the check.

```
assertions(..., Statement, AssertedStatement) :-
  Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
  get_annot(Stmts, wcet_trusted_loopbound(N), _),

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

  AssertedStatement =
    basic_block([CounterDecl,
                 while_stmt(Test, basic_block([Count|Stmts], ...), ...),
                 CounterAssert], ...).
```

Figure 3. Excerpt from the source-to-source transformer T

program. TuBound is built upon the SATIrE program analysis framework [14] and the TERMITE program transformation environment.[2] TuBound features an array of algorithms for loop analysis of different accuracy and computation cost including sophisticated analysis methods for nested loops. A detailed account of these methods can be found in [10].

3.2. Implementation

The Binary Widening/Tightening algorithms are implemented by means of a dedicated TERMITE source-to-source transformer T. For simplicity and uniformity we assume that all loops are structured. In our implementation unstructured goto-loops are thus transformed by another source-to-source transformer T' into while-loops beforehand, where possible. On while-loops the transformer T works by locating the first occurrence of a `wcet_trusted_loopbound(N)` annotation in the program source and then proceeding to rewrite the encompassing loop as illustrated in the example of Figure 3.[3] Surrounding the loop statement, a new compound statement is generated, which accommodates the declaration of a new unsigned counter variable which is initialized to zero upon entering the loop. Inside the loop, an increment statement of the counter is inserted at the very first location. After the loop, an assertion is generated which states that the count is at most of value N, where N is extracted from the annotation (cf. Figure 2).

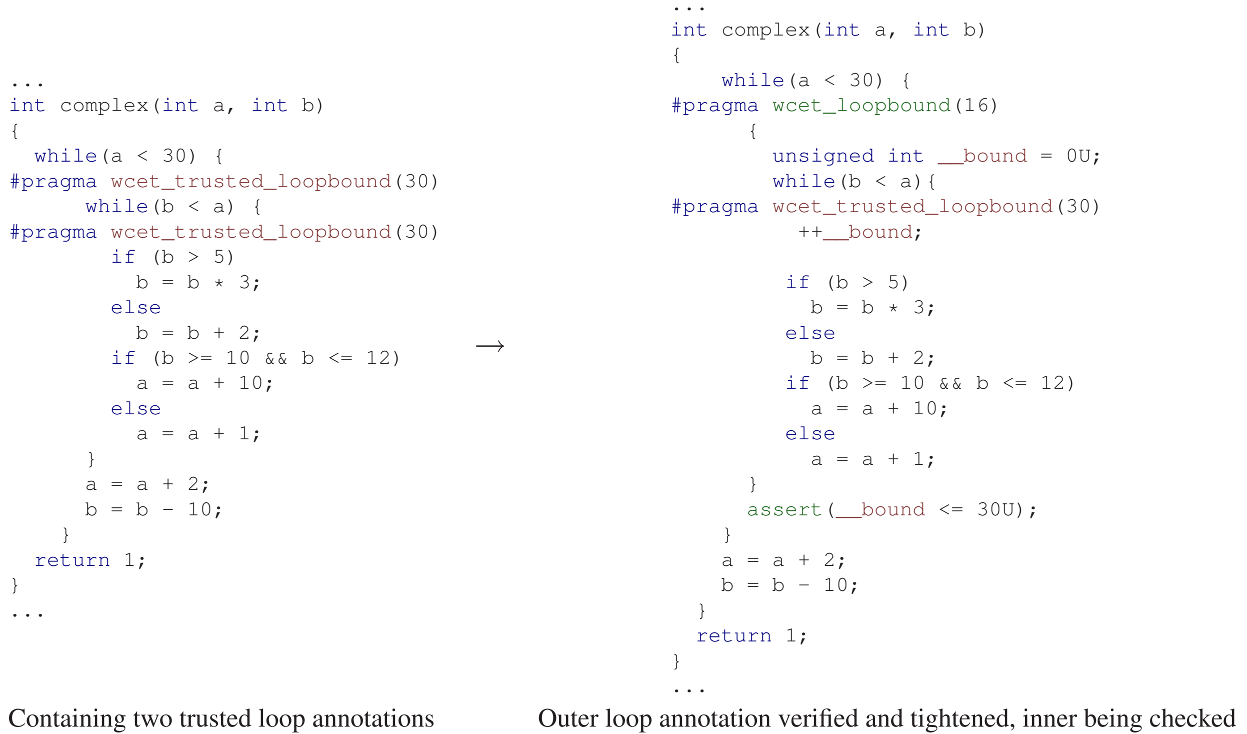
The application of the transformer is controlled by a driver, which calls the transformer for every trusted annotation contained in the source code. Depending on the result of the model checker and the coordinated application of the algorithms for binary widening and tightening, the value and the status of each annotation is updated. In the positive case, this means the status is changed from *trusted annotation* to *verified knowledge*, and the value of the originally trusted bound is replaced by a sharper, now verified bound. Figure 4 shows a snapshot of processing the *janne_complex* benchmark from the Mälardalen WCET benchmark suite. In this figure, the status and value changes are highlighted by different colors.

4. Experimental Results

We implemented our approach as an extension of the TuBound WCET analyzer and applied the extended version to the well-known Mälardalen WCET benchmark suite. As a baseline for comparison we used the 2008 version of TuBound, which took part in the 2008 WCET Tool Challenge [5]. In the

[2] `http://www.complang.tuwien.ac.at/adrian/termite`

[3] For better readability, the extra arguments containing file location and other bookkeeping information are replaced by "...".



Containing two trusted loop annotations

Outer loop annotation verified and tightened, inner being checked

Figure 4. Illustrating trusted bound verification and tightening

spirit of the WCET Tool Challenge [4, 5] we do encourage authors of other WCET analyzers to carry out similar experiments.

Our experiments conducted were guided by two questions: "Can the number of automatically bounded loops be significantly increased?" and "How expensive is the process?". The benchmarks were performed on a 3 GHz Intel Xeon processor running 64-bit Linux. The model checker used was CBMC 2.9, which we applied to testing loop bounds up to the size of $2^{13} = 8192$ using a timeout of 600 seconds, a maximum virtual memory size of 8 GiB and a maximum unroll factor of $2^{13} + 1$ [7]. The "compress" and "whet" benchmarks contained unstructured goto-loops; as indicated in Section 3.2 these were automatically converted into do-while loops beforehand by a TERMITE source-to-source transformation.

Our findings are summarized in Table 1. Column three of this table shows the percentage of loops that can be bounded by the 2008 version of TuBound; column four shows the total percentage of loops the extended version of TuBound was able to bound. The last column shows the accumulated runtime of the model checker for the remaining loops.

Comparing columns three and four reveals the superiority of the extended version of TuBound over its 2008 variant. The extended version succeeds to bound 67% of the loops the 2008 version could not bound.

Considering column five, it can be seen that the model checker terminates quickly on small problems but that the runtime and space requirements can increase to practically infeasible amounts on problems suffering from the state explosion problem. Such a behaviour can be triggered, if the initialization values which are part of the majority of the Mälardalen benchmarks are manually invalidated by introducing e.g. a faux dependency on `argc`. This demonstrates that model checking is to be

Benchmark	Loops	TuBound 2008	with Model Checking	Runtime
bs	1	0.0%	100.0%	$0.03s$
duff	2	50.0%	50.0%	$0s$
fft1	11	54.5%	81.8%	$0.43s$
janne_complex	2	0.0%	100.0%	$0.18s$
minver	17	94.1%	100.0%	$0.06s$
nsichneu	1	0.0%	100.0%	$5.59s$
qsort-exam	6	0.0%	66.6%	$0.02s$
statemate	1	0.0%	100.0%	$0.06s$
whet	11	90.9%	90.9%	$0s$
adpcm	18	83.3%	83.3%	out of memory
compress	8	25.0%	25.0%	out of memory
fir	2	50.0%	50.0%	out of memory
insertsort	2	0.0%	0.0%	out of memory
lms	10	60.0%	60.0%	out of memory
select	4	0.0%	0.0%	out of memory
bsort100	3	100.0%	100.0%	–
cnt	4	100.0%	100.0%	–
cover	3	100.0%	100.0%	–
crc	3	100.0%	100.0%	–
edn	12	100.0%	100.0%	–
expint	3	100.0%	100.0%	–
fdct	2	100.0%	100.0%	–
fibcall	1	100.0%	100.0%	–
jfdctint	3	100.0%	100.0%	–
lcdnum	1	100.0%	100.0%	–
ludcmp	11	100.0%	100.0%	–
matmult	5	100.0%	100.0%	–
ndes	12	100.0%	100.0%	–
ns	4	100.0%	100.0%	–
qurt	1	100.0%	100.0%	–
sqrt	1	100.0%	100.0%	–
st	5	100.0%	100.0%	–
recursion	0	–	–	–
Total Percentage		77.0%	84.7%	

Table 1. Results for the Mälardalen benchmarks

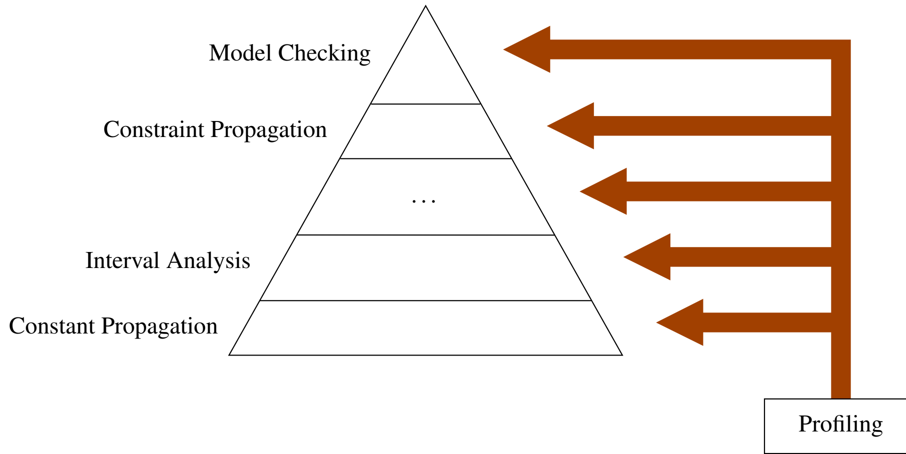


Figure 5. Pool of complementary analysis techniques with different complexity

used with care or the model checker be fed with additional information guiding and simplifying the verification task.

The fully-fledged variant of our approach, which we highlight in the next section is tailored towards this goal.

5. Extensions: The Fully Fledged Approach

The shrinking of the trusted annotation base and sharpening of time bounds, as described in Section 2, is based on model checking. Based on our experience, we believe that the model checking approach can be especially valuable in the real world when (i) it is combined with advanced program slicing techniques to reduce the state space and (ii) the results of static analyses (like TuBound's variable-interval analysis) are used to narrow the value ranges of variables, thus regaining a feasible problem size. This leads to the following extension of our approach to improve efficiency:

1. By using a pool of analysis techniques with different computational complexity: As shown in Figure 5, model checking is considered as one of the most complex analysis methods. On the other side, techniques like *constant propagation* or *interval analysis* are relatively fast. Thus we are interested in exploiting the fast techniques wherever beneficial and using the relatively complex techniques rarely.

2. By using a smart activation mechanism for the different analysis techniques: As shown on the right of Figure 5 we are interested in the interaction of the different analysis techniques. We do not aim to use the pool of analysis techniques in waves of different complexity, i.e., first applying the fast techniques and then gradually shifting towards the more complex techniques. Instead we aim for a smart interaction of the different analysis techniques. For example, whenever a technique with relatively high computational complexity has been applied, we can again apply techniques of relatively simple complexity to compute the closure of flow information based on previously obtained results; thus *squeezing* the annotation base.

We also think that profiling techniques are useful to guide the heuristics to be used within our static analysis techniques. For example, execution samples obtained by profiling can be used to elicit propositions to be verified by model checking.

The fully fledged approach envisioned in this section provides the promising potential as a research platform for complementing program analysis techniques.

6. Conclusions

Model checking has been used before in the context of WCET analyzers. For example, Metzer has used model checking to bound the WCET based on binary search [8]. In this case, the model checker provides a positive result if the proposed WCET bound is sucessfully verified. Examples of our own related work are the ForTAS [15], MoDECS [16], and ATDGEN projects [12, 13], which are concerned with measurement-based WCET analysis. In these three projects, model checking is used to generate test data for the execution of specific program paths. Intuitively, in these applications the model checker is presented with formulae stating that a specific program path is infeasible. If these formulae can be refuted by the model checker, the counter examples generated provide the test data ensuring the execution of the particular paths. Otherwise, the paths are known to be infeasible. Hence, the search for test data is in vain. In these applications the counter examples generated in the course of failing model checker runs are the truly desired output, whereas successful runs are of less interest stopping just the search for test data for the path under consideration. Similar to our approach, Rieder et al. has also used model checking to derive loop bounds [13]. They used model checking as the only approach, because of its low implementation effort. However, they conclude that model checking in general is a very costly analysis technique.

These applications of model checking are in contrast and opposite to our application of shrinking the trusted annotation base. In our application, the counter example of a failed model checker run is of little interest. We are interested in successful runs of the model checker as they allow us to change a *trusted annotation* into *verified knowledge*. To counteract the critical conclusions on model checking drawn by Rieder et al. we use model checking in concert with other analysis techniques of different complexity. This opens a new application domain for model checking in the field of WCET analysis. Our preliminary practical results demonstrate the practicality and power of this approach.

References

[1] BEYER, D., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT) 9*, 5-6 (October 2007), 505–525.

[2] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.

[3] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems 19*, 6 (1997), 992 – 1030.

[4] GUSTAFSSON, J. The WCET tool challenge 2006. In *Preliminary Proc. 2nd International IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Paphos, Cyprus, November 2006), pp. 248 – 249.

[5] HOLSTI, N., GUSTAFSSON, J., (EDS.), G. B., BALLABRIGA, C., BONENFANT, A., BOURGADE, R., CASSÉ, H., CORDES, D., KADLEC, A., KIRNER, R., KNOOP, J., LOKUCIEJEWSKI, P., MERRIAM, N., DE MICHIEL, M., PRANTL, A., RIEDER, B., ROCHANGE, C., SAINRAT, P., AND SCHORDAN, M. WCET Tool Challenge 2008: Report. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (Prague, Czech Republic, July 2008), Österreichische Computer Gesellschaft, pp. 149–171. ISBN: 978-3-85403-237-3.

[6] HORWITZ, S., REPS, T., AND SAGIV, M. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*) (1995), pp. 104 – 115.

[7] KROENING, D., AND CLARKE, E. The CPROVER User Manual. Available online at `http://www.cprover.org/cbmb/doc/manual.pdf`. (C) 2001-2008, Computer Systems Institute, ETH Zurich, Computer Science Department, Carnegie Mellon University.

[8] METZNER, A. Why model checking can improve WCET analysis. In *Proc. 16th International Conference on Computer Aided Verification* (2004), Springer, pp. 334–347. LNCS 3114.

[9] MUCHNICK, S. S. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

[10] PRANTL, A., KNOOP, J., SCHORDAN, M., AND TRISKA, M. Constraint solving for high-level wcet analysis. In *Proc. 18th International Workshop on Logic-based methods in Programming Environments (WLPE 2008)* (Udine, Italy, December 2008), pp. 77–89.

[11] PRANTL, A., SCHORDAN, M., AND KNOOP, J. TuBound – a conceptually new tool for worst-case execution time analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (Prague, Czech Republic, July 2008), Österreichische Computer Gesellschaft, pp. 141–148. ISBN: 978-3-85403-237-3.

[12] RIEDER, B. *Measurement-Based Timing Analysis of Applications written in ANSI-C*. PhD thesis, Technische Universität Wien, Vienna, Austria, Jun. 2009.

[13] 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 *Proc. 6th International Workshop on Intelligent Solutions in Embedded Systems (WISES 2008)* (Regensburg, Germany, July 2008).

[14] SCHORDAN, M. Source-To-Source Analysis with SATIrE – an Example Revisited. In *Proceedings of Dagstuhl Seminar 08161: Scalable Program Analysis* (April 2008), Germany, Dagstuhl.

[15] VIENNA UNIVERSITY OF TECHNOLOGY AND TU DARMSTADT. The ForTAS project. Web page (`http://www.fortastic.net`). Accessed in June 2009.

[16] WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. Measurement-based timing analysis. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Porto Sani, Greece, October 2008).

IS CHIP-MULTIPROCESSING THE END OF REAL-TIME SCHEDULING?

Martin Schoeberl and Peter Puschner[1]

Abstract

Chip-multiprocessing is considered the future path for performance enhancements in computer architecture. Eight processor cores on a single chip are state-of-the art and several hundreds of cores on a single die are expected in the near future. General purpose computing is facing the challenge how to use the many cores. However, in embedded real-time systems thread-level parallelism is naturally used. In this paper we assume a system where we can dedicate a single core for each thread. In that case classic real-time scheduling disappears. However, the threads, running on their dedicated core, still compete for a shared resource, the main memory. A time-sliced memory arbiter is used to avoid timing influences between threads. The schedule of the arbiter is integrated into the worst-case execution time (WCET) analysis. The WCET results are used as a feedback to regenerate the arbiter schedule. Therefore, we schedule memory access instead of CPU time.

1. Introduction

Chip-multiprocessing (CMP) is considered non-trivial for real-time systems. Scheduling of multiprocessor systems is NP-hard. Standard CMP architectures communicate via cache coherence protocols for the level 1 caches. The resolution of conflicts due to the cache coherence protocol and memory access for the cache load and write back between unrelated tasks is practically intractable for the worst-case execution (WCET) analysis of single tasks. Therefore, we consider a time-predictable CMP system with a statically scheduled access to the main memory for real-time systems [12].

In this paper, we assume following simplification: we consider a CMP system where the number of processors is equal (or higher) than the number of tasks. In such a system classic CPU scheduling becomes a non-problem as each task runs on its own CPU. What we have to consider for the feasibility analysis is the access conflict to the shared resource main memory. Access to main memory has to be scheduled between the conflicting tasks. However, scheduling the memory access is different from scheduling of tasks in a uniprocessor system: The overhead of a *context switch* in a memory arbiter is almost zero. Therefore, fine-grained scheduling algorithms can be applied at this level.

We propose a fine-grained static schedule for the memory access. We include the knowledge of the possible memory access patterns that are determined by the memory schedule into the WCET analysis. Results from the WCET analysis are fed back into the assignment of the memory schedule. Therefore, in this system scheduling and WCET analysis play an interacting role for the real-time system design and the feasibility analysis.

[1]Institute of Computer Engineering, Vienna University of Technology, Austria;
email: `mschoebe@mail.tuwien.ac.at`, `peter@vmars.tuwien.ac.at`

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2288

1.1. Background

The evaluation of the proposed system is based on following technologies: the time-predictable Java processor JOP [16], a JOP CMP system with a time-sliced memory arbiter [12], and a WCET analysis tool for embedded Java processors [20] with an extension for CMP systems.

JOP is an implementation of the Java virtual machine (JVM) in hardware. JOP was designed as a real-time processor with time-predictable execution of Java bytecodes. One feature of JOP is the so called method cache [15]. The method cache simplifies the cache analysis by caching whole methods. The accurate cycle timing of JOP enabled the development of several WCET analysis tools for embedded Java [20, 4, 3, 5]. The timing model of JOP, which is used by the WCET analysis tools, simplifies our evaluation of the memory access scheduling for a time-predictable CMP. JOP is implemented in a field-programmable gate array (FPGA). Therefore, configurations for different applications and the task sets are a valuable option.

An extension of JOP to a time-predictable chip-multiprocessor system is presented in [12]. A time-sliced memory arbitration module allows execution of several threads on different cores without influencing each other's timing. The WCET analysis tool [20] has been adapted by Pitter [13] to include the instruction timings of memory accessing bytecodes for the CMP system. The version of the WCET analysis tool we use in our evaluation is an enhanced version of [20] that includes a tighter analysis of the method cache [5].

To the best of our knowledge, the JOP CMP system is the first time-predictable CMP that includes a WCET analysis tool. Although the presented CMP architecture is based on a Java processor, the main idea of a single task per core and scheduling of memory accesses instead of threads is independent of the application language.

1.2. Related Work

Chip-multiprocessor systems are state-of-the-art in desktop and server systems and are considered also for future embedded systems. Current processors contain up to eight cores [8, 6]. The Niagara T1 from Sun [8] is a design with 8 simple six-stage, single-issue pipelines similar to the original five-stage RISC pipeline. The additional pipeline stage adds fine-grained multithreading. Using single-issue in-order pipelines is good news for WCET analysis. However, the fine-grained multithreading and communication via a shared L2 cache make WCET analysis infeasible. In contrast to the T1 our CMP system has single threaded pipelines and avoids shared caches. All communication between the cores is performed via non-cached, shared memory.

The Cell multiprocessor [6] combines, besides a standard PowerPC microprocessor, 8 synergistic processors (SP) with 256 KB local memory. The cores are connected via an on-chip communication ring [7]. The SPs are in-order pipelines and can only access their local memory. Therefore, WCET analysis of SP programs should be straight forward. The main drawback of this architecture is that the communication between the cores (and main memory) has to be performed with explicit DMA transfers. Compared to that design we allow communication via shared memory without any restrictions.

Azul Systems provides an impressive multiprocessor system for transaction oriented server workloads [2]. A single Vega chip contains 54 64-bit RISC cores, optimized for the execution of Java programs.

Up to 16 Vega processors can be combined to a cache coherent multiprocessor system resulting in 864 processors cores and supporting up to 768 GB of shared memory.

The PRET project is devoted to the development of a time-predictable CMP system [11]. The processor cores are based on the SPARC V8 instruction set architecture with a six-stage pipeline and support chip level multithreading for six threads to eliminate data forwarding and branch prediction. An interesting feature of the PRET architecture is the *deadline* instruction that stalls the current thread until a deadline is reached. This instruction is used to perform time based, instead of lock based, synchronization for access to shared data. Besides using processor local scratchpad memories, the shared memory it accessed through a so called *memory wheel*. The memory wheel is a time division, multiple access (TDMA) based memory arbiter with equal slot slice. In contrast to the PRET project, we consider TDMA schedules that can be adapted to the application needs.

The approach, which is closest related to our work, is presented in [1, 14]. The CMP system is intended for tasks according to the simple task model [9]. In this model the task communication happens explicitly: input data is read at the begin of the task and output data is written at the end of a task. Furthermore, local cache loading for the cores is performed from a shared main memory. Similar to our approach, a TDMA based memory arbitration is used. The paper deals with optimization of the TDMA schedule to reduce the WCET of the tasks. The design also considers changes of the arbiter schedule during task execution. This implies that all tasks and the cores have to be synchronized with the memory arbiter, resulting in a cyclic executive model. All tasks, even on different cores, need to fit into a common major cycle.

In contrast to [1, 14], our approach to a TDMA based CMP system allows standard shared memory communication without the restrictions of a cyclic executive. Real-time tasks with arbitrary periods are supported. The approach in [1] uses explicit path enumeration to integrate the TDMA schedule into the WCET calculation, which is known to scale poorly. In our system the TDMA schedule is integrated into an implicit path enumeration based WCET tool [12]. Furthermore, a complete implementation of the proposed CMP system and the WCET analysis tool are available.

2. WCET Based Memory Access Scheduling

Embedded systems, also called cyber-physical systems, interact with the *real* world. And the real world, external to the computer system, is massively parallel. Therefore, embedded systems are good candidates for chip-multiprocessing. We argue that the transistors required to implement super-scalar architectures are better used on complete replication of simple cores for a CMP system [19].

CMP systems share the access bandwidth to the main memory. To build a time-predictable CMP system, we need to schedule the access to the main memory in a predictable way. A predictable scheduling can only be time based, where each core receives a fixed time slice. This scheduling scheme is called time division multiple access (TDMA). The execution time of loads, stores, and the cache miss penalty depend on this schedule. Therefore, the arbiter schedule has to be known to determine accurate WCET estimates.

Assuming that enough cores are available, we propose a CMP model with a single thread per processor. In that case thread switching and schedulability analysis for each individual core disappears. Since each processor executes only a single thread, the WCET of that thread can be as long as its deadline. When the period of a thread is equal to its deadline, 100% utilization of that core is feasible.

For threads that have enough slack time left, we can increase the WCET by decreasing their share of the bandwidth on the memory bus. Other threads with tighter deadlines can, in turn, use the freed bandwidth and run faster. The consumption of the access bandwidth to main memory is adjusted by the TDMA schedule. The TDMA schedule itself is the input for WCET analysis for all threads. Finding a TDMA schedule, where all tasks meet their deadlines, is thus an iterative optimization problem.

It can be argued that the assumption of having fewer tasks than available processing cores is too restrictive. However, we do not forbid task scheduling on a single core. Our approach is still valid when more demanding tasks *own* their core and other tasks share a core under preemptive scheduling.

Two benefits of the single task per core approach have to be emphasized. First, the CPU utilization on each core can reach 100%, even when the task periods are non-harmonic, and the system is still analyzable. Second, the overhead of task dispatching, and the hard to analyze cache thrashing, disappears. Both effects lead to tighter WCET bounds and can compensate for idle cycles on the cores when the tasks are waiting on their turn to access the memory.

2.1. Memory-Access Scheduling Schemes

Before we look for a particular memory-access schedule for an application running on a multiprocessor system, we have to decide about the general properties of the scheduling scheme.

- Granularity of the memory access scheduler: one has to decide on scheduling of single memory accesses versus scheduling of longer TDMA slots comprising a larger number of memory accesses. Shorter time slices provide the highest flexibility and the most fine-grained control in distributing memory bandwidth among the processors. Longer time slices may improve access times to shared memory if larger blocks from the shared memory have to be transferred at once.
- In case time slices of multiple memory accesses are used, one further has to decide whether time slices should be of uniform duration or different length. The use of equally sized time slices provides the simplest solution. Assigning different sizes to the tasks/CPUs makes a better adaption to application needs possible.
- Single scheduling pattern versus different scheduling patterns. For some applications, the memory access characteristics of the tasks do hardly ever change or the occurrence of particular access patterns is hard to predict. Such systems may use a single TDMA memory access pattern during their whole operational cycle. Other systems, in contrast, have a number of operational phases during which the need for memory bandwidth and memory access patterns of the different tasks/CPUs differ. Such systems benefit from a more flexible memory system that supports a number of schedules tailored to the needs of the tasks in the different phases of operation.
- Synchronous versus asynchronous operation of memory-access schedules and task execution. One of the most influential decisions on the design of the system is whether the TDMA memory accesses and the task execution cycles (i.e., activation times and periods of tasks) are synchronized or not. Synchronizing memory access schedules and task operation has the benefit that memory access schedules can be optimized to the needs of the tasks during their execution cycle, at the cost of a higher planning effort and a more rigid execution pattern at runtime. Unsynchronized operation is easier to plan, but allows for a less accurate adaption of task and memory-access behavior.

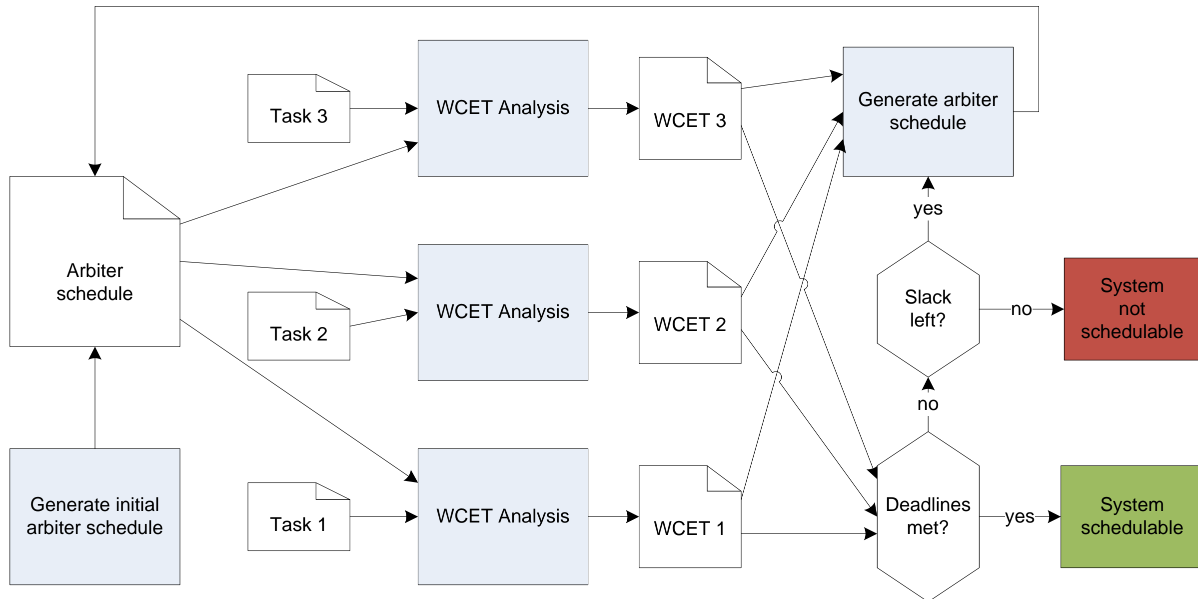


Figure 1. Tool flow for a CMP based real-time system with one task per core and a static arbiter schedule. If the deadlines are not met, the arbiter schedule is adapted according to the WCETs and deadlines of the tasks. After the update of the arbiter schedule the WCET of all tasks needs to be recalculated.

In the rest of the paper we assume that memory access schedules and task execution are not synchronized. Our goal is to find memory access schedules that split the memory bandwidth among the CPUs of the system in such a way that a maximum number of tasks of a given task set meet their timing requirements. We schedule single memory accesses and, at this time, assume that the same TDMA scheduling sequence is used throughout the whole time of system operation.

2.2. Tool Structure

Figure 1 shows the analysis tool flow for the proposed time-predictable CMP with three tasks. First, an initial arbiter schedule is generated, e.g., one with equal time slices. That schedule and the tasks are the input of WCET analysis performed for each task individually. If all tasks meet their deadlines with the resulting WCETs, the system is schedulable. If some tasks do not meet their deadline and other tasks have some slack time available, the arbiter scheduler is adapted accordingly. WCET analysis is repeated, with the new arbiter schedule, until all tasks meet their deadlines or no slack time for an adaption of the arbiter schedule is available. In the latter case no schedule for the system is found.

The TDMA schedule can be adapted along two dimensions. Firstly, the individual slot sizes can be varied. This variation can be very fine-grained down to single clock cycles. Longer slot sizes are mostly beneficial for burst transfers on a cache load. Secondly, the TDMA round can be adapted to include several (short) slots for a core within one TDMA round. This leads to lower WCET for individual memory loads and stores. Which adaption, or if a combination of the two approaches is beneficial depends on the application characteristic. In the following section both approaches are evaluated for a network application.

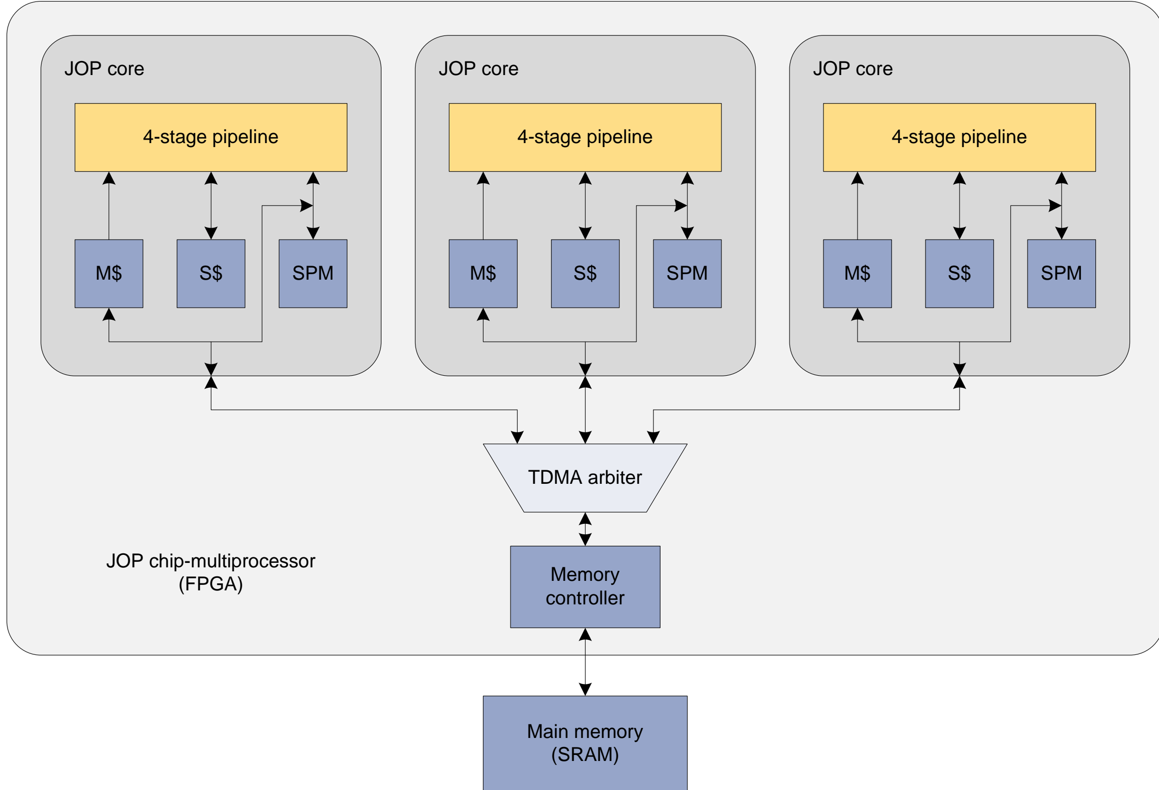


Figure 2. A JOP based CMP system with core local caches and scratchpad memories, a TDMA arbiter, the memory controller, and the shared main memory.

3. Evaluation

We evaluate our approach on the CMP version [13] of the Java processor JOP [16]. Figure 2 shows a configuration of a JOP CMP system with three cores. Each core contains a local method cache (M$), a local stack cache (S$), and a scratchpad memory (SPM). The local memories contain either thread local data (S$ and SPM) or read-only data (M$). Therefore, no cache coherency protocol is needed. The cores are connected to the TDMA based arbiter, which itself is connected to a memory controller. The memory controller interfaces to the shared, external memory.

For the WCET analysis and the schedule of the memory arbitration the memory timing is according to the implementation of the JOP CMP system on an Altera DE2 FPGA board [12]: 4 clock cycles for a 32-bit memory read and 6 clock cycles for a 32-bit memory write. The resulting minimum TDMA slot width is 6 clock cycles. The cache sizes for each core are 1 KB for the stack cache and 4 KB for the method cache.

3.1. The Baseline with the Example Application

As an example of an application with several communicating threads, we use an embedded TCP/IP stack for Java, called `ejip`. The benchmark explores the possibility of parallelization within the TCP/IP stack. The application that uses the TCP/IP stack is an artificial example of a client thread requesting a service (vector multiplication) from a server thread. That benchmark consists of 5 threads: 3 application threads (client, server, result), and 2 TCP/IP threads executing the link layer as well as the network layer protocol.

Function	Task	Single	5 cores	WCET increase
ipLink.run()	τ_1	1061	2525	2.4
resultServer()	τ_2	1526	3443	2.3
request()	τ_3	8825	23445	2.7
macServer()	τ_4	7004	17104	2.4
net.run()	τ_5	8188	18796	2.3
Sum		26604		

Table 1. WCET estimations in clock cycles for a single core and a 5 core system

One bottleneck in the original TCP/IP stack was a global buffer pool. All layers communicated via this single pool. The single pool is not an issue for a uniprocessor system, however real parallel threads compete more often for the access to the buffer pool. As a consequence, we have rewritten the TCP/IP stack to use dedicated, wait-free, single reader/writer queues [10] for the communication between the layers and the application tasks. The additional benefit of non-blocking communication is a simplification of the scheduling analysis, as we do not have to take blocking times into account.

For WCET analysis we use a new WCET tool for JOP [5] with an extension for TDMA memory arbitration [12]. The WCET tool performs the analysis at bytecode instruction level. The execution of bytecodes that access the main memory (e.g., field access, cache load on a method invoke or return) depends on the TDMA schedule. For those bytecodes, the worst-case phasing between the memory accesses and the TDMA schedule is considered for the low-level WCET timing.

Table 1 shows the WCET bounds of all 5 tasks obtained by our tool for a single core and a 5 core system connected to a TDMA arbiter with a slot size of 6 cycles for each CPU. Due to the memory access scheduling, the WCET increases for each individual task. Although 5 cores compete for the memory access the increase of the WCET is moderate: between a factor of 2.3 and 2.7. As 5 cores lead to an ideal speedup of a factor of 5 and the individual task are slower on that CMP (with respect to the analyzed WCET) by a factor of around 2.5, the net speedup of the 5 core CMP system is around a factor of $5/2.5 = 2$.

To simplify our example we assume that all tasks shall execute at the same period. As this example resembles a pipelined algorithm, this assumption is valid. With the default configuration τ_3 has the largest WCET and for a 100 MHz CMP system the minimum period is 235 µs. With this period the other tasks have a slack time between 47 µs and 210 µs. We use that slack time to adapt the TDMA schedule, as shown in Figure 1, to reduce the maximum period for all tasks. We have not yet implemented the automation of this process, but perform the arbiter schedule generation manually. The resulting WCETs for different arbiter schedules are shown in Table 2, with the equal schedule of 6 cycle slots for all cores in the second row for reference. A manual iteration of the arbiter schedule generation and WCET analysis loop took a few minutes. The WCET analysis, including data-flow analysis for loop bounds, takes around 27 s on a standard PC.

3.2. Variation of the Arbiter Schedule

For the first adaption of the arbiter schedule we assumed a quite naive approach: the slot size of the three longer tasks is doubled to 12 cycles. The result is shown in column *Schedule 1* of Table 2. The WCET for τ_4 and τ_5 has been reduced, but due to the longer TDMA round (48 cycles instead of 30

Task	Equal	Schedule 1	Schedule 2	Schedule 3	Schedule 4	Schedule 5	Schedule 6
τ_1	2525	3605	3605	5045	4325	5285	5525
τ_2	3443	4865	4865	6761	5813	7077	7393
τ_3	23445	25313	20211	20201	20473	17955	18681
τ_4	17104	15284	14860	19468	14276	16756	17376
τ_5	18796	17856	16432	14832	16204	18924	18364

Table 2. WCET estimations in clock cycles of the application tasks with different arbiter configurations

cycles) the WCET of task τ_3 actually increased – an effect into the wrong direction.

For *Schedule 2* and *Schedule 3* the basic slot size was kept to the minimum of 6 cycles, but different inter-leavings of the tasks are used: $< \tau_1, \tau_3, \tau_4, \tau_5, \tau_2, \tau_3, \tau_4, \tau_5 >$ and $< \tau_3, \tau_3, \tau_5,\ \tau_5, \tau_4,\ \tau_1, \tau_3, \tau_3,$ $\tau_5, \tau_5, \tau_4, \tau_2 >$. For *Schedule 3* the repetition of two 6 cycle slots for the same task results effectively in a single 12 cycle slot. Both schedules decrease the execution time of task τ_3 to 202 µs at the expense of higher execution times of some other tasks.

The following experiments are based on *Schedule 2* with variations of the individual slot sizes. We use following notation for the schedule: τ_i/n is a slot of n cycles for task i. Schedule 4, $< \tau_1/6, \tau_3/8, \tau_4/8, \tau_5/8, \tau_2/6, \tau_3/8, \tau_4/8, \tau_5/8 >$ increases the slot size to 8 cycles for all, but tasks τ_1 and τ_2. The resulting minimum period, 205 µs, is similar to *Schedule 2*. To reduce the WCET of τ_3 we double the slot size for τ_3, resulting in Schedule 5, $< \tau_1/6, \tau_3/16, \tau_4/8,\ \tau_5/8, \tau_2/6,\ \tau_3/16, \tau_4/8,$ $\tau_5/8 >$, with a minimum period, now determined by task τ_5, of 189 µs. After several additional iterations we ended up with *Schedule 6*, $< \tau_1/6, \tau_3/16, \tau_4/8, \tau_5/10, \tau_2/6, \tau_3/16, \tau_4/8, \tau_5/10 >$, providing the best balanced system for the three demanding tasks and a minimum period of 187 µs. Any further increase of individual slot sizes resulted in a higher execution time for τ_3, τ_4, and τ_5. Therefore, not the whole slack time of τ_1 and τ_2 could be distributed to the other three tasks.

3.3. Discussion

The decrease of the minimum period for all tasks from 235 µs to 187 µs due to the optimization of the memory arbiter schedule is probably not too exciting. However, we started our experiment with a task set that is already quite balanced: three tasks had a similar WCET, and only two tasks had a shorter WCET. The resulting slack time of two tasks can be distributed to the other three tasks.

An interesting question is how the WCET performance scales with a CMP system with a TDMA based memory arbitration? As we assumed the same period for all tasks we can perform a simple comparison: when executing the five tasks on a single core the cumulative WCET is the sum of the WCET values as given in Table 1: 26604, resulting in a minimum period of the system of 266 µs. The same task set on the 5 core CMP system with the optimized arbiter schedule results in a minimum period of 187 µs. Therefore, the speedup of the WCET due to a 5 core CMP is about 1.4. This speedup is moderate, but it ignores more complex task sets with different periods and the overhead of preemptive scheduling on a single core, which is not present in the single task per core setup.

This paper presents a first step towards scheduling of memory accesses instead of tasks. The WCET analysis considers the arbiter schedule only for memory accesses within a single bytecode. An improvement of the analysis that considers basic blocks for the interleaving of memory accesses with

the TDMA schedule can provide tighter WCET bounds.

With respect to the TDMA schedule we are currently restricted to quite regular schedules. An extension to irregular patterns is straight forward and will enable a finer tuning of the memory bandwidth requirements.

Furthermore, we have avoided caching of heap data to simplify the predictability of the system. As a result, each access to heap allocated data has a WCET of a complete TDMA round, which easily reaches cache miss latencies of current architectures. First ideas on time-predictable data caching emerge [18]. A small, fully associative data cache will enable analysis of some memory accesses to heap allocated data. Even a low analyzable hit rate of the data cache will improve the WCET of the tasks on the CMP system.

We are confident that the suggested improvements will reduce the WCET of tasks on a CMP system considerable and enable usage of time-predictable CMP systems in hard real-time systems.

4. Conclusion

In this paper, we presented the idea to build chip-multiprocessor real-time systems where each application task has its own CPU for execution. Therefore, task scheduling is completely avoided. The remaining shared resource, the main memory and the memory bandwidth, needs to be scheduled. A time-slicing arbitration of the memory access is the only time-predictable form for CMP systems. When the arbitration schedule is known in advance the worst-case memory access time is known and can be integrate into WCET analysis.

Furthermore, we have presented an iterative approach to build the schedule for the memory arbiter. Starting from equal sized time slices for each core/task a first WCET estimation for each task is calculated. Tasks that have a shorter WCET than their deadline can be punished by reducing their share of the memory bandwidth. That bandwidth can be used by tasks that do not (yet) meet their deadline. The updated arbitration schedule is again used as input for WCET analysis. We have demonstrated this iterative approach with an example application in the context of a Java processor based CMP system.

The processor design, the TDMA based CMP design, and the WCET analysis tool are open source under the GNU GPL. The sources are available via git[2] and the build instructions can be found in [17].

Acknowledgements

We thank Christof Pitter for the design and implementation of the TDMA arbiter and the integration into the WCET tool during his PhD thesis. Furthermore, we are thankful to Benedikt Huber who has redesigned the WCET analysis tool for JOP during his Master's thesis. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD) and 214373 (Artist Design).

[2] `git clone git://www.soc.tuwien.ac.at/jop.git`

References

[1] ANDREI, A., ELES, P., PENG, Z., AND ROSEN, J. Predictable implementation of real-time applications on multiprocessor systems on chip. In *Proceedings of the 21st Intl. Conference on VLSI Design* (Jan. 2008), pp. 103–110.

[2] AZUL. Azul compute appliances. Whitepaper, 2009.

[3] BOGHOLM, T., KRAGH-HANSEN, H., OLSEN, P., THOMSEN, B., AND LARSEN, K. G. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)* (New York, NY, USA, 2008), ACM, pp. 106–114.

[4] HARMON, T. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[5] HUBER, B. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[6] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. J. Introduction to the Cell multiprocessor. *j-IBM-JRD 49*, 4/5 (2005), 589–604.

[7] KISTLER, M., PERRONE, M., AND PETRINI, F. Cell multiprocessor communication network: Built for speed. *Micro, IEEE 26* (2006), 10–25.

[8] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (2005), 21–29.

[9] KOPETZ, H. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.

[10] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[11] LICKLY, B., LIU, I., KIM, S., PATEL, H. D., EDWARDS, S. A., AND LEE, E. A. Predictable programming on a precision timed architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)* (Atlanta, GA, USA, October 2008), E. R. Altman, Ed., ACM, pp. 137–146.

[12] PITTER, C. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)* (Santa Clara, USA, September 2008), ACM Press, pp. 115–122.

[13] PITTER, C. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[14] ROSEN, J., ANDREI, A., ELES, P., AND PENG, Z. 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)* (Dec. 2007), pp. 49–60.

[15] SCHOEBERL, M. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)* (Agia Napa, Cyprus, October 2004), vol. 3292 of *LNCS*, Springer, pp. 371–382.

[16] SCHOEBERL, M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2* (2008), 265–286.

[17] SCHOEBERL, M. *JOP Reference Handbook*. No. ISBN 978-1438239699. CreateSpace, 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.

[18] SCHOEBERL, M. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)* (Tokyo, Japan, March 2009), IEEE Computer Society.

[19] SCHOEBERL, M. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems vol. 2009, Article ID 758480* (2009), 17 pages.

[20] SCHOEBERL, M., AND PEDERSEN, R. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)* (New York, NY, USA, 2006), ACM Press, pp. 202–211.

MAKING DYNAMIC MEMORY ALLOCATION STATIC TO SUPPORT WCET ANALYSES[1]

Jörg Herter[2] and Jan Reineke[2]

Abstract

Current worst-case execution time (WCET) analyses do not support programs using dynamic memory allocation. This is mainly due to the unpredictable cache performance when standard memory allocators are used. We present algorithms to compute a static allocation for programs using dynamic memory allocation. Our algorithms strive to produce static allocations that lead to minimal WCET times in a subsequent WCET analyses. Preliminary experiments suggest that static allocations for hard real-time applications can be computed at reasonable computational costs.

1. Introduction

High cache predictability is a prerequisite for precise worst-case execution time (WCET) analyses. Current static WCET analyses fail to cope with programs that dynamically allocate memory mainly because of their unpredictable cache behavior. Hence, programmers revert to static allocation for hard real-time systems. However, sometimes dynamic memory allocation has advantages over static memory allocation. From a programmer's point of view, dynamic memory allocation can be more natural to use as it gives a clearer program structure. In-situ transformation of one data structure into another one may reduce the necessary memory space from the sum of the space needed for both structures to the space needed to store the larger structure.

But why do WCET analyses fail to cope with dynamic memory allocation? In order to give safe and reasonably precise bounds on a program's worst-case execution time, analyses have to derive tight bounds on the cache performance, i.e., they have to statically classify the memory accesses as hits or misses. Programs that dynamically allocate memory rely on standard libraries to allocate memory on the heap. Standard `malloc` implementations, however, inhibit a static classification of memory accesses into cache hits and cache misses. Such implementations are optimized to cause little fragmentation and neither provide guarantees about their own execution time, nor do they provide guarantees about the memory addresses they return. The cache set that memory allocated by `malloc` maps to is statically unpredictable. Consequently, WCET analyses cannot predict cache hits for accesses to dynamically allocated memory. Additionally, dynamic memory allocators pollute the cache themselves. They manage free memory blocks in internal data structures which they maintain and traverse during (de)allocation requests. An unpredictable traversal of these data structures results in an equally unpredictable influence on the cache.

While it seems that dynamic memory allocation prohibits a precise WCET analysis, the special re-

[1]This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), the German-Israeli Foundation (GIF) in the "Encasa" project, and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).
[2]Universität des Saarlandes, Saarbrücken, Germany, {jherter, reineke}@cs.uni-saarland.de

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2284

quirements for hard real-time systems can be utilized to circumvent predictability issues introduced by dynamic allocation. Hard-real time software contains no unbounded loops nor unbounded recursion and hence no unbounded allocation. In this work, we show how information necessary for WCET analysis on hard real-time systems (like the aforementioned loop and recursion bounds) can be used to transform dynamic memory allocation into static memory allocation. Our proposed algorithm statically determines a memory address for each dynamically allocated heap object such that

- the WCET bound calculated by the standard IPET method is minimized,
- as a secondary objective the memory consumption is minimized, and
- no objects that may be contemporaneously allocated overlap in memory.

We evaluate our approach on several academic example programs.

The following subsection gives an overview on related work, while Section 2 illustrates the importance of cache predictability for WCET analyses and the problems introduced by dynamic memory allocation. In Section 3, we propose algorithms to compute static allocations from program descriptions to substitute dynamic allocation by. Experimental results are given in Section 4, further improvements of our approach are sketched in Section 5. Section 6 concludes the paper.

1.1. Related Work

There are two other approaches to make programs that dynamically allocate memory more analyzable with respect to their WCET. In [4] we proposed to utilize a predictable memory allocator to overcome the problems introduced by standard memory allocators. Martin Schoeberl proposes different (hardware) caches for different data areas [10]. Hence, accesses to heap allocated objects would not influence cached stack or constant data. The cache designated for heap allocated data would be implemented as a fully-associative cache with an LRU replacement policy. For such an architecture it would be possible to perform a cache analysis without knowledge of the memory addresses of the heap allocated data. However, a fully-associative cache, in particular with LRU replacement, cannot be very big, due to technological constraints.

2. Caches, Cache Predictability, and Dynamic Memory Allocation

Caches are used to bridge the increasing gap between processor speeds and memory access times. A cache is a small, fast memory that stores a subset of the contents of the large but slow main memory. It is located at or near the processor. Due to the *principle of locality*, most memory accesses can be serviced by the cache, although it is much smaller than the main memory. This enables caches to drastically improve the average latency of memory accesses. To reduce cache management and data transfer overhead, the main memory is logically partitioned into a set of *memory blocks* of size b. Memory blocks are cached as a whole in *cache lines* of equal size. When accessing a memory block, the system has to determine whether the memory block is currently present in the cache or if it needs to be fetched from main memory. To enable an efficient look-up, each memory block can be stored in a small number of cache lines only. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set, i.e., the number of cache lines it consists of, is called the *associativity* k of the cache. At present, k ranges from 1 to 32. Since the number of memory blocks that map to a set

is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. *Cache analyses* [2] strive to derive tight bounds on the cache performance. To obtain such bounds, analyses have to classify memory accesses as cache hits or cache misses. A memory access constitutes a *cache hit* if it can be served by the cache. And analogously, *cache miss* denotes a memory access that cannot be served by the cache; instead the requested data has to be fetched from main memory. The better the classification of accesses as cache hits or cache misses is, the tighter are the obtained bounds on the cache performance. To classify memory accesses, a cache analysis needs to know the mapping of program data to cache sets. Otherwise, it does not know which memory blocks compete for the cache lines of each cache set. Using standard dynamic memory allocators, like for instance [7], no knowledge about the mapping of allocated data structures to cache sets is statically available. Assume, for example, a program would allocate 6 memory blocks to hold objects of a singly-linked list. Two possible mappings from cache sets to those objects, assuming a 4-way cache-associative cache with 4 cache sets, are given in Figure 1 (a) and (b), respectively.

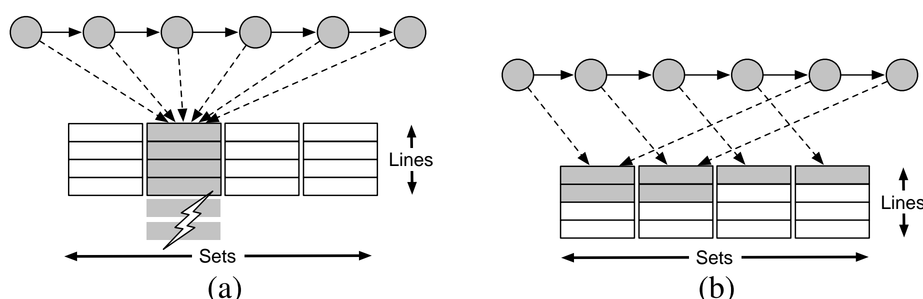


Figure 1. Two possible cache mappings of 6 dynamically allocated objects organized in a singly-linked list.

For the mapping in Figure 1 (a) a further traversal of the list would result in no cache hits at all. Given the mapping depicted in Figure 1 (b), a subsequent traversal of the list would result in 6 cache hits. A conservative cache analysis with no knowledge about the addresses of allocated memory has no other option than to classify all accesses to allocated memory as cache misses, while during actual program runs potentially all accesses may be cache hits. In modern processors, turning off the cache can easily cause a thirty-fold increase in execution time [6], hence, conservatively treating all accesses as cache misses yields very imprecise and thus useless analysis results.

There is a second problem due to dynamic memory allocators besides the resulting inability to guarantee cache hits for the dynamically allocated data. Due to the unpredictable mapping of such data to cache sets, knowledge derived about the caching of statically allocated data is lost if dynamically allocated data is accessed. As it is not known to which cache set newly allocated memory maps to, the analysis has to conservatively treat such memory as potentially mapping to every cache set and evicting cached content from these sets.

Also the manner in which dynamic allocators determine free memory addresses significantly decreases cache predictability. Unused memory blocks are managed in some internal data structure or data structures by the allocators. Upon allocation requests, these structures are traversed in order to find a suitable free block. This traversal is in general statically unpredictable and leads to an equally unpredictable cache pollution as all memory blocks visited during traversal are loaded into the cache. As it is unpredictable which and also how many free blocks are considered upon an allocation request, the response time of the request is also not predictable. Only upper bounds with potentially high overestimations can be given for the execution time of a call of the memory allocation procedure. For a

WCET analysis that relies on tight execution time bounds this is highly undesirable.

In summary, WCET analyses face three problems when dynamic memory allocation is employed: (1) the mapping to cache sets of allocated data is unpredictable, (2) malloc itself causes unpredictable cache pollution, and (3) no bounds on the response times of malloc are guaranteed.

3. Precomputing Static Memory Addresses

To avoid the described problems with dynamic memory allocators, we propose to utilize available knowledge about the program used in a WCET analysis to substitute dynamic memory allocation by static allocation. For hard real-time applications, a good memory mapping has to enable the computation of small WCET bounds.

What do we mean by a memory mapping? We partition the main memory into *memory blocks* the size of a cache line, s.t. each block maps into exactly one cache set, i.e., the memory blocks are aligned with the cache sets. As we will later address the memory at the granularity of memory blocks and never below, *memory address* $(i-1)$ refers to the i-th memory block. Analogously to the main memory, we also partition the program's dynamically allocated memory into blocks the size of a cache line which we call *heap allocated objects*. A memory mapping assigns to each heap allocated object a memory block.

A heap allocated object may contain several or just a fraction of a single program object(s). If a program object does not fit into a single heap allocated object, it has to be spread over several. Heap allocated objects holding a single program object have to be placed consecutively in memory. To this end, we introduce *heap allocated structures*: ordered sets of heap allocated objects. The heap allocated objects of a heap allocated structure shall be mapped to consecutive memory addresses.

We can relate heap allocated structures either to dynamically allocated program objects, like for example single nodes of a binary search tree, or to entire data structures built from dynamically allocated objects, like for example a linked list consisting of several node objects. What relation we choose depends on the program we want to analyze. For example, it may be advantageous to consider a linked list structure to be a single heap allocated structure, while for larger objects organized in a tree structure we might want to consider each single object a heap allocated structure.

Hence, formally, a memory mapping m is a function that maps each heap allocated memory object to a memory address: $m = \bigcup_{o_{i,j} \in \mathcal{O}} \{o_{i,j} \mapsto a_{i,j}\}$ where $o_{i,j} \in \mathcal{O}$ is a heap allocated object, $a_{i,j}$ its memory address, and $(i,j) \in \mathcal{I} \times \mathcal{J}_i$ an index for elements of $\mathcal{O}$, the set of all heap allocated objects. Furthermore, $\mathcal{I}$ denotes an index set for the set of heap allocated structures and, for all $i \in \mathcal{I}$, $\mathcal{J}_i$ an index set for the set of (heap allocated) objects of structure i.

What we want to compute for a program P is a memory mapping that allows for a WCET bound on P of

$$\min_{\substack{\text{memory}\\ \text{mapping } m}} \text{WCET}(P, m)$$

by using as little memory as possible.

WCET Computation by IPET How can we compute WCET(P, m)? As proposed by Li and Malik, the longest execution path of a program can be computed by implicit path enumeration techniques (IPET) [8]. Their approach formulates the problem of finding the longest execution path as an integer linear program (ILP). Given the control-flow graph $G(P) = (V, E, s, e)$ of a program P and loop bounds b_k for the loops contained in P, this ILP is constructed as follows. First, we introduce two types of counter variables: x_i—the execution frequency of basic block B_i—for counting the number of executions of basic block $B_i \in V$ and y_j for storing the number of traversals of edge $E_j \in E$. We can then represent and describe the possible control flow by stating that the start and end node of the control-flow graph are executed exactly once (1). Each node is executed as often as the control flow enters and leaves the node ((2) and (3)). And finally, equation (4) incorporates loop bounds into our ILP representation of the program's control flow. For each loop l with an (upper) iteration bound of b_l, we add a constraint ensuring that each outgoing edge of the first block b within the loop is taken at most as often as the the sum over all ingoing edges to b times the loop bound b_l.

$$x_i = 1 \quad \text{if} \quad B_i = s \vee B_i = e \tag{1}$$

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (\cdot, B_k)\} \tag{2}$$

$$\sum_{j \in \mathcal{J}} y_j = x_k \quad \text{where} \quad \mathcal{J} = \{j \mid E_j = (B_k, \cdot)\} \tag{3}$$

$$y_l \leq b_l \cdot \left(\sum_{j \in \mathcal{J}} y_j \right) \quad \text{where} \quad \mathcal{J} = \{j \mid E_j \text{ is loop entry edge to } l\} \tag{4}$$

The WCET bound is then obtained using the objective function:

$$\max \sum_{i \in \{i \mid B_i \in V\}} x_i c_i^m$$

where c_i^m is an upper bound on the WCET of basic block B_i for a given memory mapping m.

WCET-optimal Memory Mapping A WCET-optimal memory mapping yields a WCET bound equal to

$$\min_{\substack{\text{memory} \\ \text{mapping } m}} \max \sum_{i \in \{i \mid B_i \in V\}} x_i c_i^m \tag{5}$$

However, this problem is no ILP anymore.
We propose the following heuristic approach to compute approximate solutions to (5). Initially, we start with a memory mapping m that uses minimal memory. We then compute the WCET of the program for the memory mapping m using an IPET as described above. An improved mapping can then be obtained by selecting the basic block B_i whose penalty due to conflict misses has the greatest contribution to the WCET bound and modifying m such that c_i^m is minimized. The last step can be repeated until no further improvements can be achieved.
Algorithm 1 implements this strategy as a hill climbing algorithm that internally relies on methods to

compute a bound on the WCET of a program using an IPET approach ($\text{WCET}_{\text{IPET}}()$), to compute an initial memory optimal mapping ($mem_opt()$), and to compute a block optimal mapping for a given basic block ($block_opt()$). To enable our hill climbing algorithm to escape local maxima, we allow for a certain number of side steps. With side steps, we mean that the algorithm is allowed to select an equally good or even worse mapping if no improved mapping can be found during an iteration. The maximum number of allowed side steps should be at least the number of program blocks in order to allow the optimization of each block. In our experiments, we set the maximum number of side steps to $2 \cdot |\mathcal{B}|$, where $|\mathcal{B}|$ is the number of program blocks.

An ILP formulation to implement $\text{WCET}_{\text{IPET}}()$ has already been introduced. We can also implement $mem_opt()$ and $block_opt()$ as ILPs as follows.

Memory Optimal Mapping Let $a_{i,j}$ be an integer variable of an ILP storing the memory address of the j-th heap allocated object of heap allocated structure i. Again, with memory address, we do not mean the physical address, but rather the $a_{i,j}$-th memory block by partitioning the memory into blocks of the size of a cache line. For all i, j, i', j' s.t. the j-th object of structure i may be allocated at the same time as the j'-th object of structure i', we add two constraints:

$$a_{i',j'} + 1 - a_{i,j} - b_{i,j,i',j'} \cdot \mathcal{C} \leq 0 \quad (6)$$
$$a_{i',j'} - (a_{i,j} + 1) + (1 - b_{i,j,i',j'}) \cdot \mathcal{C} \geq 0 \quad (7)$$

where the $b_{a,b,c,d}$ are auxiliary binary variables and $\mathcal{C}$ a constant larger than the value any expression within the ILP can take. Such a $\mathcal{C}$ may be computed statically. These constraints ensure that parts of structures allocated at the same time do not reside at the same memory address. To enforce a consecutive placement of parts of the same structure, we add for all i, j

$$a_{i,j+1} = a_{i,j} + 1 \quad (8)$$

Computation of the largest used memory address $\bar{a}$ is done by the following constraints for all i, j:

$$a_{i,j} < \bar{a} \quad (9)$$

We want to minimize our memory consumption, hence we use the following objective function in order to minimize the largest address in use:

$$\min \bar{a} \quad (10)$$

Algorithm 1: Hill climbing algorithm to compute a suitable memory mapping

Data: functions $\text{WCET}_{\text{IPET}}()$, $mem_opt()$, and $block_opt()$; $sideSteps$ number of allowed side steps
Result: near WCET-optimal memory mapping for dynamically allocated objects

$mapping_{best} \leftarrow mem_opt()$;
$mapping_{curr} \leftarrow mapping_{best}$;
while $sideSteps > 0$ **do**
    calculate $\text{WCET}_{\text{IPET}}(mapping_{curr})$;
    select basic block b with largest WCET contribution due to conflict misses;
    $mapping_{tmp} \leftarrow block_opt(mapping_{curr}, b)$;
    if $WCET_{IPET}(mapping_{tmp}) < WCET_{IPET}(mapping_{curr})$ **then**
        $mapping_{curr} \leftarrow mapping_{tmp}$;
        if $WCET_{IPET}(mapping_{tmp}) < WCET_{IPET}(mapping_{best})$ **then**
            $mapping_{best} \leftarrow mapping_{tmp}$;
        end
    else
        $sideSteps \leftarrow sideSteps - 1$;
        $mapping_{curr} \leftarrow mapping_{tmp}$;
    end
end

Block Optimal Mapping To get a block optimal memory mapping with respect to potential conflict cache misses, we modify the ILP used to compute a memory usage optimal mapping as follows. We first compute the cache sets to which memory blocks are mapped that are accessed in the basic block we want to optimize the mapping for.

Let $\#cs$ denote the number of cache sets and $cs_{a,i,j}$ denote binary variables set to 1 iff the j-th block of structure i is mapped to cache set a. The following set of constraints (for each a, i, j, s.t. a is a cache set and block j of structure i is accessed in the considered basic block) sets these $cs_{a,i,j}$ accordingly. However, several auxiliary variables have to be set previously. Let $o_{i,j}$ denote the j-th block of structure i. Then $a_{i,j}$ stores the cache set to which $o_{i,j}$ is mapped, and $n_{i,j}$ stores the result of an integer division of the address of $o_{i,j}$ by $\#cs$. Furthermore, $y_{a,i,j}$ is set to $|a - cs_{i,j}|$ and $ltz_{a,i,j}$ to 0 iff $a - cs_{i,j} < 0$.

$$a_{i,j} - n_{i,j} \cdot \#cs - cs_{i,j} = 0 \tag{11}$$
$$n_{i,j} \geq 0 \tag{12}$$
$$cs_{i,j} \geq 0 \tag{13}$$
$$cs_{i,j} - \#cs + 1 \leq 0 \tag{14}$$
$$a - cs_{i,j} - ltz_{a,i,j} \cdot \mathcal{C} \leq 0 \tag{15}$$
$$a - cs_{i,j} + (1 - ltz_{a,i,j}) \cdot \mathcal{C} \geq 0 \tag{16}$$
$$a - cs_{i,j} \leq y_{a,i,j} \tag{17}$$
$$-(a - cs_{i,j}) \leq y_{a,i,j} \tag{18}$$
$$a - cs_{i,j} + (1 - ltz_{a,i,j}) \cdot \mathcal{C} \geq y_{a,i,j} \tag{19}$$
$$-(a - cs_{i,j}) + ltz_{a,i,j} \cdot \mathcal{C} \geq y_{a,i,j} \tag{20}$$

Finally, we assign $cs_{a,i,j}$ using the following constraints:

$$y_{a,i,j} - \mathcal{C} \cdot (1 - cs_{a,i,j}) \leq 0 \tag{21}$$

$$y_{a,i,j} - (1 - cs_{a,i,j}) \geq 0 \tag{22}$$

We compute the number of potential cache misses p_a resulting from accessing cache set a using the following constraints for all cache sets a; b_a being auxiliary binary variables, and k denoting the associativity of the cache:

$$\left(\sum_{i,j} cs_{a,i,j} \right) - k - b_a \cdot \mathcal{C} \leq 0 \tag{23}$$

$$\left(\sum_{i,j} cs_{a,i,j} \right) - k + (1 - b_a) \cdot \mathcal{C} \geq 0 \tag{24}$$

$$0 \leq p_a \tag{25}$$

$$\left(\sum_{i,j} cs_{a,i,j} \right) - (1 - b_a) \cdot \mathcal{C} \leq p_a \tag{26}$$

The objective function itself is replaced by

$$\min \; \mathcal{C} \cdot \sum_{0 \leq a < k} p_a + \bar{a} \tag{27}$$

with the intention of minimizing the number of conflict misses that may occur during execution of the considered basic block. However, if the number of cache sets increases, the ILP to compute the block optimal mapping becomes intractable independently of the number of heap objects. Experiments suggest that for target hardware with 256 or more cache sets solutions to the ILPs cannot be computed in reasonable time anymore. Computing block optimal mappings using ILP formulations also introduces severe complexity issues for programs with a large number of heap allocated memory blocks. In order to enable the analysis of larger programs and/or programs for hardware with more cache sets, we propose to replace the block optimal ILP by a heuristic algorithm. Consider a simulated annealing [5] algorithm as sketched in Algorithm 2. The neighborhood $N(m)$ of a memory mapping m can be defined as the set of all memory mappings, s.t. the address of exactly 1 structure s differs in its address in m by 1. Formally speaking, for a memory mapping $m = \bigcup_{o_{i,j} \in \mathcal{O}} \{o_{i,j} \mapsto a_{i,j}\}$, the neighborhood is defined as $N(m) = \left\{ m \oplus \{o_{i,j} \mapsto a'_{i,j}\} \middle| |a_{i,j} - a'_{i,j}| = 1 \right\}$. The evaluation function to determine the costs of a memory mapping m is defined as

$$eval(m) = \mathcal{C}^2 \cdot memoryConflicts(m) + \mathcal{C} \cdot potentialConflictMisses(m) + maxAddr(m)$$

The first component ensures that memory mappings without memory conflicts are preferred. The number of potential conflict misses is minimized by the second component. For mappings with the same number of conflict misses, component three favors the ones with minimal memory consumption. As maximal temperature we use $Temperature_{MAX} = |\mathcal{O}|^2$. The intention is to have $Temperature_{MAX}$ large enough to allow for almost random behavior at the beginning of each restart, converging to a (local) optimum. The current cooling ratio is $1.2^{-|\mathcal{I}| \cdot restarts}$.

Although, a memory optimal mapping can be computed by our ILP formulation for most programs, this procedure can be easily replaced by a similar simulated annealing heuristic.

Algorithm 2: Simulated annealing algorithm to compute a suitable block optimal mapping

Result: memory mapping with minimal potential conflict misses for program block b

$restarts \leftarrow 0$;
$mapping_{best} \leftarrow mem_opt()$;
while $restarts < RESTARTS$ **do**
    $mapping_{curr} \leftarrow mem_opt()$;
    $Temperature \leftarrow Temperature_{MAX}$;
    while $Temperature > Temperature_{MIN}$ **do**
        compute the set $N(mapping_{curr})$ of neighboring mappings of $mapping_{curr}$;
        foreach $mapping_{tmp} \in N(mapping_{curr})$ **do**
            if $eval(mapping_{tmp}) < eval(mapping_{best})$ **then**
                $mapping_{best} \leftarrow mapping_{tmp}$;
            end
            set $mapping_{curr}$ to $mapping_{tmp}$ with probability $e^{\frac{eval(mapping_{curr})-eval(mapping_{tmp})}{Temperature}}$;
        end
        $Temperature \leftarrow Temperature \cdot Cooling_Ratio$;
    end
    $restarts \leftarrow restarts + 1$;
end

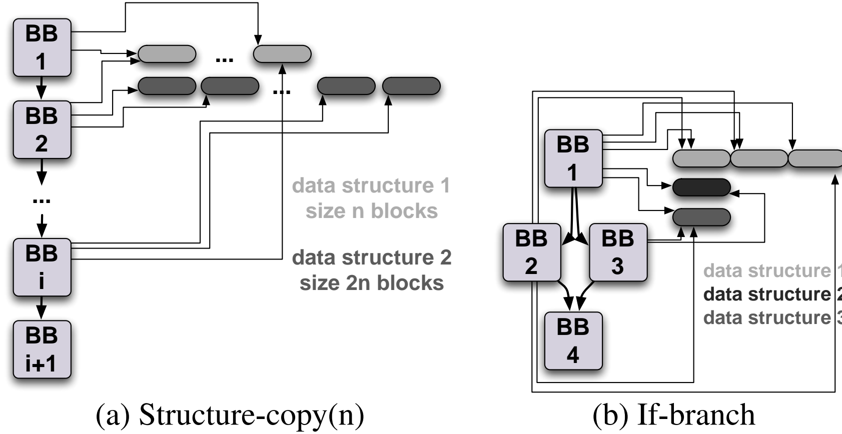
4. Experiments

Our preliminary experiments consider as target platforms the PowerPC MPC603e and a simpler toy architecture (DTA): a 2-way set-associative cache with 2 cache sets. The PowerPC CPU utilizes separated 4-way set-associative instruction and data caches of 16 KB, organized in 128 cache sets with a cache line size of 32 bytes.

Unfortunately, we lack a collection of real-life programs on which we could evaluate our algorithm. We therefore consider a typical textbook example used to motivate the use of dynamic memory allocation. When transforming one data structure into another, dynamic memory allocation can lead to a memory consumption smaller than the sum of the memory needed to hold both structures by freeing each component of the first data structure as soon as its contents are copied to the second structure. Suppose you have a data structure A consisting of n objects and each object occupies l memory blocks. How much memory is needed to transform this structure into a structure B with n objects each occupying k memory blocks? Assuming that $k \geq l$, a moments thought reveals that we need $n \cdot k + l$ memory blocks for this transformation. We tried our algorithm on several program descriptions, where data structures consisting of n heap objects are transformed into other structures consisting of an equal number of objects of larger size (Structure-copy(n)). In all instances, the computed solution used $n \cdot k + l$ memory blocks. In Structure-copy(3)ex we added an additional program block to Structure-copy(3) in which the first and the last block of the first structure as well as the last block of the second structure are accessed. On the simplified hardware the memory optimal mapping for this program is not WCET-optimal anymore. The new WCET-optimal memory uses one additional memory address to resolve the potential conflict misses in the additional program block. Our algorithm successfully discovered that.

As a more complex example, consider a program as sketched in Figure 2 (b). Here, allocating all 5

blocks sequentially is memory and WCET-optimal for the PPC hardware. However, the best memory mapping for our toy hardware would be to allocate the memory block for data structure 3 before that of structure 2. Again, our algorithm discovers that.



(a) Structure-copy(n) (b) If-branch

Figure 2. (Basic-)Block Graph with data accesses for some example programs.

Program	(Target) Hardware	Memory-/Block-Optimal Algorithm	Analysis Time (min/max/**avg**) ms
Structure-copy(3)	DTA	ILP/ILP	23/64/**38.5**
Structure-copy(3)	DTA	ILP/Simulated Annealing	22/82/**49.6**
Structure-copy(3)	PPC603e	ILP/ILP	23/79/**43.2**
Structure-copy(3)	PPC603e	ILP/Simulated Annealing	23/89/**43.6**
Structure-copy(3)ex	DTA	ILP/ILP	73/164/**100.1**
Structure-copy(3)ex	DTA	ILP/Simulated Annealing	29/52/**35.7**
Structure-copy(3)ex	PPC603e	ILP/ILP	23/32/**24.9**
Structure-copy(3)ex	PPC603e	ILP/Simulated Annealing	23/30/**24.8**
If-branch	DTA	ILP/ILP	43,332/44,005/**43,755**
If-branch	DTA	ILP/Simulated Annealing	26/35/**28.5**
If-branch	PPC603e	ILP/ILP	17/41/**20.7**
If-branch	PPC603e	ILP/Simulated Annealing	17/25/**18.4**
Structure-copy(100)	DTA	Simulated Annealing/Simulated Annealing	9,118/9,710/**9,362.3**
Structure-copy(100)	PPC603e	Simulated Annealing/Simulated Annealing	9,025/9,629/**9,345.1**
Structure-copy(500)	PPC603e	Simulated Annealing/Simulated Annealing	260,401/276,797/**268,932.7**

Figure 3. Running times of example analyses each executed 10 times on a 2.66 GHz Core 2 Duo with 2 GB RAM.

5. Improvements and Future Work

Our algorithms rely on solving ILPs whose complexity might significantly increase with the number of data structures or more precisely with the number of $a_{i,j}$ variables. However, many programs allocate objects on the heap, although these objects do not survive the function frame they are allocated in. Hence, such heap objects might be allocated on the stack, in which case they would not contribute to the complexity of our algorithm. Escape analysis is a static analysis that determines which heap allocated objects' lifetimes are not bounded by the stack frame they are allocated in [1, 3]. By allocating objects on the stack that do not escape the procedure they are created in, 13% to 95% of a programs heap data can be moved onto the stack [1]. Incorporating an escape analysis and a program transformation moving heap objects onto the stack might therefore significantly reduce the complexity of our algorithm for real-life programs. Therefore, we are currently developing a novel, precise escape analysis based on shape analysis via 3-valued logic [9].

6. Conclusions

Transforming dynamic allocation into static allocation circumvents the main problems of WCET analyses introduced by the unpredictability of memory allocators. Preliminary benchmarks and experiments suggest that such a transformation can be computed with reasonable efforts. However, whether it is feasible to fully automate the process of transforming dynamic allocation to static allocation, i.e., also automatically generate program descriptions, and at what computational costs still remains to be seen.

References

[1] BLANCHET, B. Escape Analysis for Object-Oriented Languages: Application to Java, *SIGPLAN Not. Vol. 10*, 1999.

[2] FERDINAND, C. and WILHELM, R. Efficient and Precise Cache Behavior Prediction for Real-Time Systems, *Real-Time Systems*, 17(2-3):131–181, 1999.

[3] GAY, D. and STEENSGAARD, B. Fast Escape Analysis and Stack Allocation for Object-Based Programs, *CC'00*, 82–93, 2000.

[4] HERTER, J., REINEKE, J., and WILHELM, R. CAMA: Cache-Aware Memory Allocation for WCET Analysis, *WiP Session of ECRTS'08*, 24–27, July 2008.

[5] KIRKPATRICK, S., GELATT, C.D., and VECCHI, M.P. Optimization by Simulated Annealing, *Science, Number 4598*, 671–680, 1983.

[6] LANGENBACH, M., THESING, S., and HECKMANN, R. Pipeline Modeling for Timing Analysis, *Proceedings of the Static Analyses Symposium (SAS)*, volume 2477, 2002.

[7] LEA, D. A Memory Allocator, *Unix/Mail, 6/96*, 1996.

[8] LI, Y.S. and MALIK, S. Performance Analysis of Embedded Software Using Implicit Path Enumeration, *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 456–461, 1995.

[9] SAGIV, M., REPS, T., and WILHELM, R. Parametric Shape Analysis via 3-valued Logic, *ACM Transactions on Programming Languages and Systems, Vol. 24, No. 3*, Pages 217–298, May 2002.

[10] SCHOEBERL, M. Time-predictable Cache Organization, *STFSSD'09*, March 2009.

Invited talk

PREDICTABLE IMPLEMENTATION OF REAL-TIME APPLICATIONS ON MULTIPROCESSOR SYSTEMS ON CHIP

Petru Eles[1]

Abstract

Worst-case execution time (WCET) analysis and, in general, the predictability of real-time applications implemented on multiprocessor systems has been addressed only in very restrictive and particular contexts. One important aspect that makes the analysis difficult is the estimation of the system's communication behavior. The traffic on the bus does not solely originate from data transfers due to data dependencies between tasks, but is also affected by memory transfers as result of cache misses. As opposed to the analysis performed for a single processor system, where the cache miss penalty is constant, in a multiprocessor system each cache miss has a variable penalty, depending on the bus contention. This affects the tasks' WCET which, however, is needed in order to perform system scheduling. At the same time, the WCET depends on the system schedule due to the bus interference. In this context, we present an approach to worst-case execution time analysis and system scheduling for real-time applications implemented on multiprocessor SoC architectures. We will also address the bus scheduling policy and its optimization, which are of huge importance for the performance of such predictable multiprocessor applications.

1 Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2287

SOUND AND EFFICIENT WCET ANALYSIS IN THE PRESENCE OF TIMING ANOMALIES[1]

Jan Reineke[2] and Rathijit Sen[3]

Abstract

Worst-Case-Execution-Time (WCET) analysis computes upper bounds on the execution time of a program on a given hardware platform. Abstractions employed for static timing analysis can lead to non-determinism that may require the analyzer to evaluate an exponential number of choices even for straight-line code. Pruning the search space is potentially unsafe because of "timing anomalies" where local worst-case choices may not lead to the global worst-case scenario. In this paper we present an approach towards more efficient WCET analysis that uses precomputed information to safely discard analysis states.

1. Introduction

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints. Timeliness of reactions is absolutely necessary. Off-line guarantees on the worst-case execution time of each task have to be derived using safe methods.

Static worst-case execution time (WCET) tools employ abstract models of the underlying hardware platforms. Such models abstract away from parts of the hardware that do not influence the timing of instructions, like e.g. the values of registers. In addition to the abstraction of data, such models further abstract components like branch predictors and caches, that have a very large state space. Good abstractions of hardware components can predict their precise concrete behavior most of the time. However, in general, abstractions introduce non-determinism: whenever the abstract model cannot determine the value of a condition, it has to consider all possibilities, to cover any possible concrete behavior.

Considering all possibilities can make timing analysis very expensive. Even on straight-line code the timing analyzer might have to consider an exponential number of possibilities in the length of the path. Often, the different possibilities can be intuitively classified as local worst- or local best-cases: e.g. cache miss vs. cache hit, pipeline stall or not, branch misprediction or not, etc. In such cases, it is tempting to only follow the local worst-case if one is interested in the WCET. Unfortunately, due to *timing anomalies* [12, 15] this is not always sound. A timing anomaly is a situation where the local worst-case does not entail the global worst-case. For instance, a cache miss—the local worst-case—may result in a shorter execution time, than a cache hit, because of scheduling effects. See Figure 1 for an example. Shortening task A leads to a longer overall schedule, because task B can

[1]This work has profited from discussions within the ARTIST2 Network of Excellence. It is supported by the German Research Foundation (DFG) as part of SFB/TR AVACS and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

[2]Universität des Saarlandes, Saarbrücken, Germany, reineke@cs.uni-saarland.de

[3]University of Wisconsin, Madison, USA, rathijit@cs.wisc.edu

now block the "more" important task C, which may only run on Resource 2. Analogously, there are cases where shortening a task leads to an even greater decrease in the overall schedule. Reineke et al. [15] formally define timing anomalies and give a set of examples for timing anomalies involving caches and speculation.

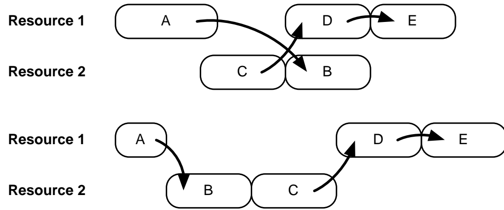


Figure 1. Scheduling Anomaly.

In the presence of timing anomalies it is unsound to simply discard non local-worst-cases. We assume that most if not all modern processors exhibit timing anomalies and that there is little hope to construct processors with good performance that do not exhibit them. To obtain an efficient timing analysis, we would still like to discard non-local-worst-case states, yet in a sound way. Our idea is to perform a precomputation on the abstract model of the hardware, that is used for timing analysis. The precomputation computes for each pair of analysis states a bound on the future difference in timing between the two states. This precomputation might be very expensive, but it only has to be done once for a particular abstract hardware model. Timing analyses using this hardware model can then safely discard analysis states based on the precomputed differences, even in the presence of timing anomalies. Assuming that timing anomalies occur seldomly and can be excluded using the precomputation most of the time, the proposed analysis will be much more efficient than previous exhaustive methods.

2. Related Work

Graham [7] shows that a greedy scheduler can produce a longer schedule, if provided with shorter tasks, fewer dependencies, more processors, etc. Graham also gives bounds on these effects, which are known as scheduling anomalies today.

Lundqvist & Stenström first introduced timing anomalies in the sense relevant for timing analysis. In [12] they give an example of a cache miss resulting in a shorter execution time than a cache hit. A timing anomaly is characterized as a situation where a positive (negative) change of the latency of the first instruction results in a global decrease (increase) of the execution time of a sequence of instructions. Situations where the local effect is even accelerated are also considered timing anomalies, i.e. the global increase (decrease) of the execution time is greater than the local change.

In his PhD thesis [3] and a paper with Jonsson [4], Engblom also discusses timing anomalies. He translates the notion of timing anomalies of the Lundqvist/Stenström paper [12] to his model by assuming that single pipeline stages take longer, in contrast to whole instructions. Both Lundqvist and Engblom claim that, in processors that contain in-order resources only, no timing anomalies can occur. This is not always true unfortunately, as corrected in Lundqvist's thesis [11]. Schneider [17] and Wenzel et al. [21] note that if there exist several resources that have overlapping, but unequal capabilities, timing anomalies can also occur.

Reineke et al. [15] provide the first formal definition of timing anomalies. The definition of timing anomalies in this paper is a slight relaxation of that of [15]. In recent work, Kirner et al. [9] introduce the notion of parallel timing anomalies. Such anomalies arise if a WCET analysis is split into the analysis of several hardware components that operate in parallel. Depending on how the analysis

results are combined the resulting WCET estimate may be unsafe. The authors identify conditions that guarantee the safety of two combination methods. This work is orthogonal to ours.

3. Static WCET Analysis Framework

Over the last several years, a more or less standard architecture for static timing-analysis tools has emerged [8, 19, 5]. Figure 2 gives a general view on this architecture. One can distinguish three major building blocks:

1. Control-flow reconstruction and static analyses for control and data flow.
2. Micro-architectural analysis, which computes upper and lower bounds on execution times of basic blocks.
3. Global bound analysis, which computes upper and lower bounds for the whole program.

Micro-architectural analysis [6, 11, 3, 20, 10, 16] determines bounds on the execution time of basic blocks, taking into account the processor's pipeline, caches, and speculation concepts. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc. Ignoring these average-case-enhancing features would result in very imprecise bounds.

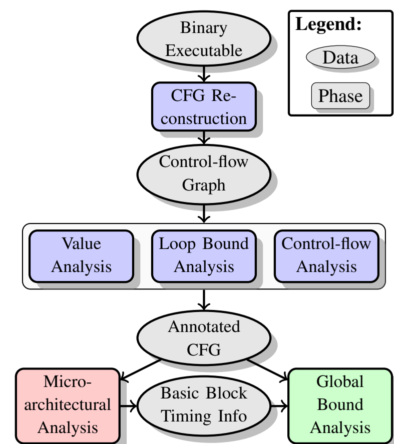


Figure 2. Main components of a timing-analysis framework and their interaction.

In this paper, we consider micro-architectural analyses that are based on abstractions of the concrete hardware and that propagate abstract states through the control-flow-graph [3, 20, 6]. In such analyses, the micro-architectural analysis is the most expensive part of the WCET analysis. Due to the introduction of non-determinism by abstraction and the possibility of *timing anomalies* it is necessary to consider very many cases. For complex architectures, this may yield hundreds of millions of analysis states, and may thus be very memory and time intensive.

In the following, we will describe a simple, formal model of such a micro-architectural analysis. Later, we will describe how this model can be extended to enable more efficient analyses.

3.1. A formal model of micro-architectural analysis

Micro-architectural analysis determines bounds on the execution times of basic blocks. Using an abstract model of the hardware, it "simulates" the possible behavior of the hardware in each cycle. Due to uncertainty about the concrete state of the hardware or about its inputs, abstract models are—unlike usual cycle-accurate simulators—non-deterministic. Therefore, an abstract state may have several possible successor states.

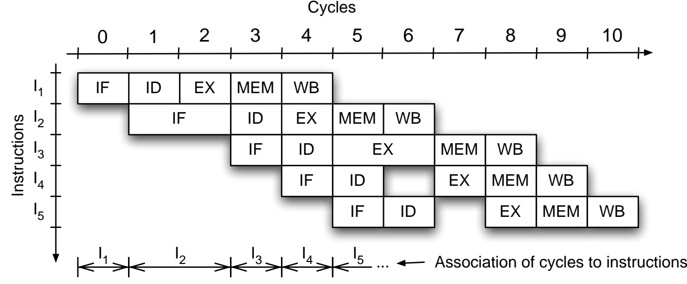


Figure 3. Pipelined execution of several instructions and association of instructions with execution cycles (bottom).

Cycle semantics. Let $State$ be the set of states of the abstract hardware model, and let $Prog$ be the set of programs that can be executed on the hardware. Then the *cycle semantics* can be formally modeled by the following function:

$$cycle : State \times Prog \rightarrow \mathcal{P}(State).$$

It takes an abstract state and a program and computes the set of possible successor states. Such an abstract hardware model can be proved correct by showing that it is an abstract interpretation [2] of the concrete hardware model. To bound the execution time of a basic block, one needs to associate execution cycles with instructions in the program. In pipelined processors several instructions are executed simultaneously, see Figure 3. There is no canonical association of execution cycles to instructions. One of several possibilities is to associate a cycle with the last instruction that was fetched. This is exemplified in the lower part of Figure 3.

Instruction semantics. Based on such a connection between execution cycles and instructions, one can lift the cycle semantics to an instruction semantics. This instruction semantics takes an abstract hardware state and an instruction that is to be fetched and computes the set of possible abstract hardware states until the next instruction can be fetched. Each of the resulting abstract hardware states is associated with the number of cycles to reach this state:

$$exec_I : State \times I \rightarrow \mathcal{P}(State \times \mathbb{N}).$$

For a formalization of the connection of the two semantics, see [20]. As an example of the two semantics in the analysis of the execution of a basic block, see Figure 4.

Instruction semantics can be easily lifted to the execution of basic blocks, which are simply sequences of instructions $\in I^*$:

$$exec_{BB} : State \times I^* \rightarrow \mathcal{P}(State \times \mathbb{N})$$

$$\begin{aligned} exec_{BB}(s, \epsilon) &:= \{(s, 0)\} \\ exec_{BB}(s, \iota_0 \dots \iota_n) &:= \{(s'', t' + t'') \mid \ (s', t') \in exec_I(s, \iota_0) \wedge \\ &\qquad (s'', t'') \in exec_{BB}(s', \iota_1 \dots \iota_n)\} \end{aligned}$$

As a shortcut, we will also write $s \xrightarrow[\iota]{t'} s'$ for $(s', t') \in exec_I(s, \iota)$ and similarly $s \xrightarrow[\iota_0 \cdots \iota_n]{t'} s'$ for $(s', t') \in exec_{BB}(s, \iota_0 \dots \iota_n)$.

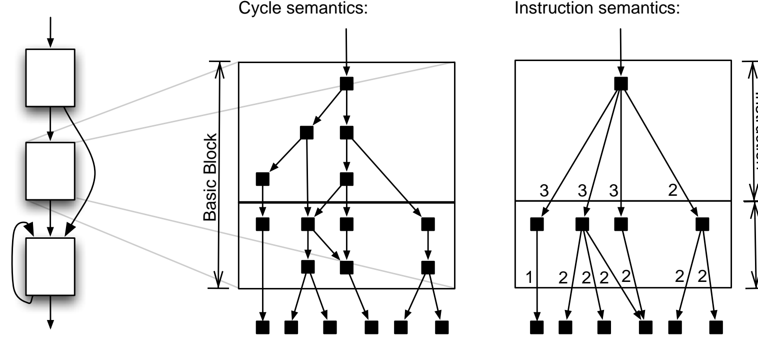


Figure 4. Cycle semantics and instruction semantics of a basic block.

Bounds on the execution times of basic blocks. Given an instruction semantics on a finite set of states $State$, one can compute bounds on the execution times of the basic blocks of a program. To that end, one can compute the set of abstract states that might reach the start of each basic block through a fixed-point iteration. For each of the states that might reach a basic block, the instruction semantics can then be used to compute the maximum execution time of the block starting in that state:

$$\max(s, \iota_0 \dots \iota_n) := \max\{t \mid s \xrightarrow[\iota_0 \dots \iota_n]{t} s'\}. \tag{1}$$

Finally, the maximum of these times for each state reaching a basic block is an upper bound on the execution time of that basic block in the given program. Analogously to the computation of upper bounds one can also compute lower bounds on the execution time of a basic block by the $\min$ function:

$$\min(s, \iota_0 \dots \iota_n) := \min\{t \mid s \xrightarrow[\iota_0 \dots \iota_n]{t} s'\}. \tag{2}$$

4. Timing Anomalies, Domino Effects, and How to Safely Discard States

The approach described in the previous section can be very expensive. Due to non-determinism introduced by abstraction, the set of states to be considered can be extremely large. It is therefore tempting to discard states that are non local-worst-case, e.g., states resulting from a cache hit, if both a cache hit and a cache miss may happen.

4.1. Timing anomalies

Unfortunately, due to so-called *timing anomalies* [12] this is not always sound. The following is a definition of timing anomalies that is slightly relaxed compared with that of [15].

Definition 1 (Timing anomaly)**.** *An instruction semantics has a* timing anomaly *if there exists a sequence of instructions $\iota_0\iota_1 \dots \iota_n \in I^*$, and an abstract state $s \in State$, such that*

- *there are states $s_1, s_2 \in State$, with $s \xrightarrow[\iota_0]{t_1} s_1$ and $s \xrightarrow[\iota_0]{t_2} s_2$, and $t_1 < t_2$, such that*
- $t_1 + \max(s_1, \iota_1 \dots \iota_n) > t_2 + \max(s_2, \iota_1 \dots \iota_n)$.

In an instruction semantics with timing anomalies it is unsound to discard a non-local-worst-case state (like s_1 in the definition). The execution of non-local-worst-case states may be so much slower than the execution of local-worst-case states, that the non-local-worst-case state yields the global worst-case timing. Experience shows that reasonable abstract instruction semantics for most modern processors exhibit timing anomalies.

4.2. How to safely discard analysis states

Our approach is to precompute a function $\Delta : State \times State \to \mathbb{N}^\infty$, where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$, that bounds the maximal difference in worst-case timing between the two states on any possible instruction sequence. Sometimes the difference in timing between two states cannot be bounded by any constant. That is why we augment $\mathbb{N}$ with ∞. The order of natural numbers is lifted to $\mathbb{N}^\infty$ by adding $\infty \geq n$ for all $n \in \mathbb{N}^\infty$.

Definition 2 (Valid Δ)**.** *A Δ function is* valid*, if for all pairs of states $s_1, s_2 \in State$ and for all instruction sequences $\iota_0 \dots \iota_n \in I^*$:*

$$\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \dots \iota_n) - \max(s_2, \iota_0 \dots \iota_n).$$

Given such a Δ function it is possible to safely discard analysis states. If the analysis encounters two states s_1 and s_2 with execution times t_1 and t_2, respectively, it may discard s_2 if $t_1 - t_2 \geq \Delta(s_2, s_1)$ and s_1 if $t_2 - t_1 \geq \Delta(s_1, s_2)$. In these cases, the discarded state can never overtake the other state. So Δ can be used to locally exclude the occurrence of a timing anomaly. It is expected that this is often the case, as timing anomalies are not the common case. This optimization does not influence the precision of the analysis.

Even if $t_1 - t_2 < \Delta(s_2, s_1)$, one can safely discard s_2, by adding the penalty $\Delta(s_2, s_1) - (t_1 - t_2)$ to t_1. For the resulting $t_1' = t_1 + \Delta(s_2, s_1) - (t_1 - t_2)$, it holds that $t_1' - t_2 \geq \Delta(s_2, s_1)$. In contrast to the first optimization, this of course comes at the price of decreased precision. This optimization offers a way of trading precision for efficiency.

4.3. Domino effects

Unfortunately, the difference in timing between two states cannot always be bounded by a constant. This situation is known as a *domino effect*[1].

Definition 3 (Domino effect)**.** *An instruction semantics has a* domino effect *if there are two states $s_1, s_2 \in State$, such that for each $\Delta \in \mathbb{N}$ there is a sequence of instructions $\iota_0 \dots \iota_n \in I^*$, such that*

$$\max(s_1, \iota_0 \dots \iota_n) - \max(s_2, \iota_0 \dots \iota_n) \geq \Delta.$$

In other words, there are two states whose timing may arbitrarily diverge. Such effects are known to exist in pipelines [17] and caches [1, 14]. If such domino effects exist for many pairs of states, valid Δ functions will often have value ∞. In that case, the Δ function is rather useless; it can rarely be used to discard states.

[1] Domino effects are also known as *unbounded timing effects* [11].

Although the difference in execution times may not be bounded by a constant, the ratio between the two execution times is always bounded, i.e., $\frac{\max(s_1,\iota_0\ldots\iota_n)}{\max(s_2,\iota_0\ldots\iota_n)} < \rho$ for some constant ρ. An alternative to computing Δ functions is to compute two functions $\rho : State \times State \rightarrow \mathbb{Q}$ and $\delta : State \times State \rightarrow \mathbb{Q}$ that together bound the maximal difference between the two states in the following way:

Definition 4 (Valid ρ and δ functions)**.** *A pair of functions ρ and δ is* valid*, if for all pairs of states $s_1, s_2 \in State$ and instruction sequences $\iota_0 \ldots \iota_n \in I^*$:*

$$\max(s_1, \iota_0 \ldots \iota_n) \leq \rho(s_1, s_2) \cdot \max(s_2, \iota_0 \ldots \iota_n) + \delta(s_1, s_2).$$

If there are no domino effects, there are valid ρ and δ functions in which ρ is 1 everywhere. In that case, δ is valid in the sense of Definition 2. Otherwise, ρ and δ can still be used to safely discard states: Say the analysis wants to discard a state s_1 with execution time t_1, but keep another state s_2 with execution time t_2, s.t. $\rho(s_1, s_2) = 1.05$ and $\delta(s_1, s_2) = 5$. Then, the analysis could discard s_1, but remember to multiply the future execution time of s_2 by 1.05. Also, similarly to the case of Δ functions, if $t_2 - t_1 < 5$, it would have to add $5 - (t_2 - t_1)$ to t_2.

5. Computation of Valid Δ Functions

Given an instruction semantics, how to compute a valid Δ function? Of course, one cannot simply enumerate all sequences of instructions. However, it is possible to define a system of recursive constraints whose solutions are valid Δ functions. These recursive equations correspond to the execution of no instruction at all or to the execution of a single instruction. Therefore, a finite number of constraints will suffice.

As Δ needs to bound the difference in timing for all instruction sequences, it must do so in particular for the empty sequence. This implies the constraints

$$\Delta(s_1, s_2) \quad \geq \quad 0 \tag{3}$$

for all $s_1, s_2 \in State$. Longer sequences can be covered by the recursive constraints

$$\Delta(s_1, s_2) \quad \geq \quad t_1' - t_2' + \Delta(s_1', s_2') \tag{4}$$

for all $s_1, s_2, s_1', s_2' \in State$ such that $s_1 \xrightarrow[\iota]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota]{t_2'} s_2'$ for some ι.

Theorem 1. *Any solution Δ to this set of constraints is a valid Δ function.*

Proof. A Δ function that satisfies the above constraints actually fulfills a stronger condition than validity. The constraints guarantee that $\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \ldots \iota_n) - \min(s_2, \iota_0 \ldots \iota_n)$ for all $s_1, s_2 \in State$ and $\iota_0 \ldots \iota_n \in I^*$. Proof by induction over the length of the sequence $\iota_0 \ldots \iota_n$:

Base case: We have to show that $\Delta(s_1, s_2) \geq \max(s_1, \epsilon) - \min(s_2, \epsilon)$. Since $\max(s_1, \epsilon) - \min(s_2, \epsilon) = 0$, this is trivially fulfilled by satisfaction of the $\Delta(s_1, s_2) \geq 0$ constraints.

Inductive step: We have to show that $\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \ldots \iota_{n+1}) - \min(s_2, \iota_0 \ldots \iota_{n+1})$ given that $\Delta(s_1, s_2) \geq \max(s_1, \iota_1 \ldots \iota_{n+1}) - \min(s_2, \iota_1 \ldots \iota_{n+1})$. The recursive constraints guarantee that

$$\Delta(s_1, s_2) \geq \max\{t_1' - t_2' + \Delta(s_1', s_2') \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota_0]{t_2'} s_2'\}.$$

Plugging the induction hypothesis (I.H.) for $\Delta(s_1', s_2')$ into this yields

$$\begin{aligned}
&\Delta(s_1, s_2) \\
\overset{\text{I.H.}}{\geq}\ & \max\{t_1' - t_2' + \max(s_1', \iota_1 \ldots \iota_{n+1}) - \min(s_2', \iota_1 \ldots \iota_{n+1}) \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota_0]{t_2'} s_2'\} \\
\overset{\text{Eq. 1,2}}{=}\ & \max\{t_1' - t_2' + t_1'' - t_2'' \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_1' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_1''} s_1'' \wedge\ s_2 \xrightarrow[\iota_0]{t_2'} s_2' \wedge s_2' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_2''} s_2''\} \\
=\ & \max\{t_1' + t_1'' \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_1' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_1''} s_1''\} - \min\{t_2' + t_2'' \mid s_2 \xrightarrow[\iota_0]{t_2'} s_2' \wedge s_2' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_2''} s_2''\} \\
\overset{\text{Eq. 1,2}}{=}\ & \max(s_1, \iota_0 \ldots \iota_{n+1}) - \min(s_2, \iota_0 \ldots \iota_{n+1})
\end{aligned}$$

□

The lower the function value of Δ, the more states can be discarded using Δ. We are therefore computing the *least* solution to the set of constraints.

Our constraints fall into the class of *difference constraints*. The least solution of a system of difference constraints can be found by solving a shortest path problem [13]. To compute this efficiently, one can use Tarjan's algorithm of subtree disassembly and FIFO selection rule [18]. Negative cycles in the constraint graph correspond to domino effects. Tarjan's algorithm allows for an efficient detection and elimination of these cycles. The least solution for pairs of states on these cycles is ∞.

We have also implemented a solution to compute valid ρ and δ functions, which allows to bound the effect of domino effects. However, due to space constraints, we cannot present the theoretical background and the results of these computations here.

6. Case Study

We used the analysis technique described above on a processor model that includes a decoupled fetch pipeline, multi-cycle functional units and variable memory access latencies. Figure 5 shows a diagrammatic representation of this model. `E0` through `Ek` represent functional units that execute instructions. A particular instruction may be scheduled to be executed on any one of a subset of the available functional units. The `L/S` unit is for load-store instructions only. Our model currently does not consider delays due to reorder-buffer unavailability or bus contention cycles. However, stalls due to data dependencies, functional unit occupancy, and limited fetch buffer space are considered. Speculative execution is not modeled.

Specifications of instances of this model are used as inputs to our analysis engine that computes the Δ function as described in Section 5 A simple architecture description language is used to describe the processors. The size of the state space is reduced by exploiting symmetries in the specification. For example, instructions are classified into types according to their execution latencies. The analyzer first builds a transition system corresponding to this specification. Each state in the transition system comprises of the states of the functional units, fetch buffers, and fetch and data accesses in transit. The edge between any two adjacent states, S_1 and S_2 in this transition system is annotated with the pair $(t, \mathtt{I})$ that indicates a transition from state S_1 to S_2 after t cycles due to execution of instruction `I`. Once we have built this transition system we can construct the constraint system described in the previous section in a straightforward way.

Figure 6 shows part of the transition system for a processor with a domino effect. In this processor, functional unit `E0` is optimized for instruction type `I1` whereas `E1` is optimized for `I2`. `I1` needs 2

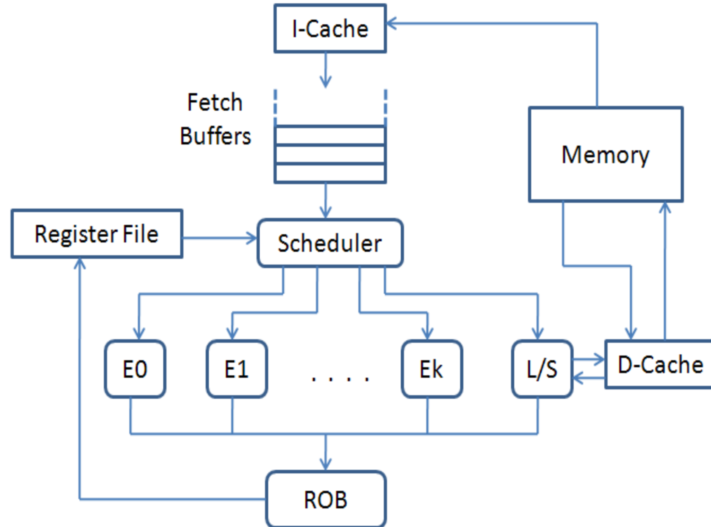


Figure 5. Processor Architecture

cycles to execute on `E0`, but 4 cycles on `E1`. The timing requirements for `I2` are opposite. There are 2 fetch buffers that are drained in order and Icache hit time is 1 cycle. In the figure, `Ij:m` indicates that instruction type `Ij` is in its $\texttt{m}^{th}$ cycle of execution on that functional unit. A shaded functional unit indicates a new instruction dispatch (`m`=1) that cycle. The figure shows that the timing difference between paths starting from states S_0 and S_4 is 1 clock cycle per loop. The timing difference is thus unbounded on the instruction sequence (`I1.I2`)*. It is useful to use the cycle ratio (4/3) in this case.

The transition system for a simple processor example with 2 instruction types, 2 functional units, execution times ranging from 2 to 6 cycles, 4 fetch buffers and no domino effects had 555 states and generated 97340 constraints. The Δ function ranged from 0 through 7 with 0s forming 88.1% of the distribution. Currently we have not experimented with full specifications of real processors. Our preliminary investigations with scaled down models of processors suggest that the constraint graphs are usually sparse: the number of inequations for most of the toy models typically lie well within 0.1% of the theoretical bound of $|S|^4$, where S is the set of states in which a new instruction is fetched.

7. Conclusions and Future Work

We presented a simple model of micro-architectural analysis which resembles several existing analyses. Timing anomalies in abstract hardware models prevent sound *and* efficient micro-architectural analysis. To enable efficient and yet sound analysis, we introduced Δ functions, which allow to safely prune analysis states even for architectures that exhibit timing anomalies. We have shown how to compute valid Δ functions for a hardware model and evaluated our approach on two example architectures. We have also introduced valid ρ and δ functions, which allow to prune states even in the presence of domino effects. On the way, we arrived at a slightly simpler definition of timing anomalies than in [15] and the first formal definition of domino effects.

In future work, we plan to apply our approach to real architectures, like the MOTOROLA POWERPC 75X or the ARM9, and evaluate the improvement in analysis efficiency. We have seen that valid Δ functions also allow to trade precision for additional efficiency. It will be interesting to study this trade-off on real architectures.

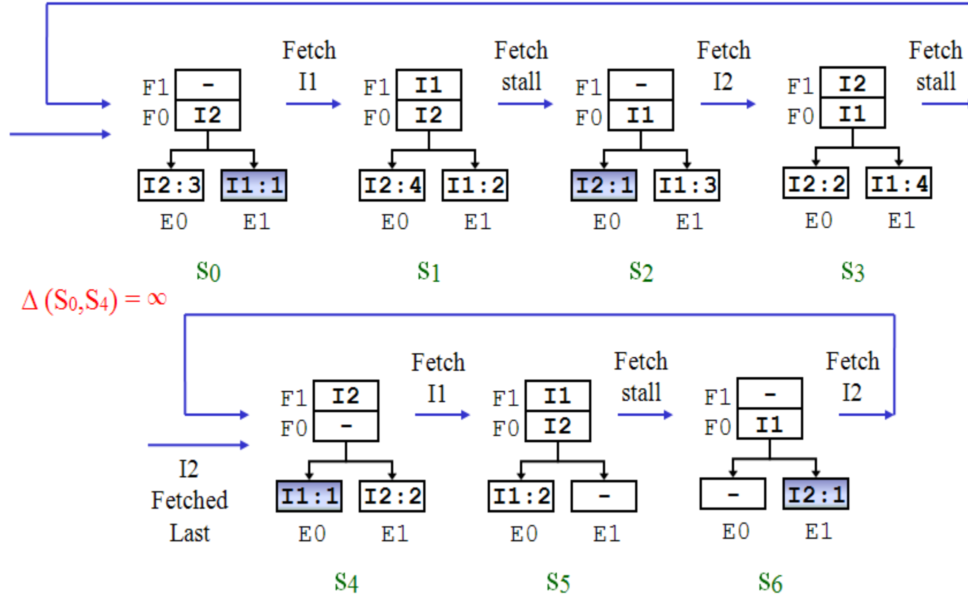


Figure 6. Example with Domino Effects

Acknowledgements The authors would like to thank Daniel Grund and Claire Burguière for valuable comments on drafts of this paper.

References

[1] BERG, C. PLRU cache domino effects. In *WCET '06* (2006), Schloss Dagstuhl, Germany.

[2] COUSOT, P., AND COUSOT, R. *Building the Information Society*. Kluwer Academic Publishers, 2004, ch. Basic Concepts of Abstract Interpretation, pp. 359–366.

[3] ENGBLOM, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[4] ENGBLOM, J., AND JONSSON, B. Processor pipelines and their properties for static WCET analysis. In *EMSOFT'02* (London, UK, 2002), Springer-Verlag, pp. 334–348.

[5] ERMEDAHL, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[6] FERDINAND, C., AND WILHELM, R. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems 17*, 2-3 (1999), 131–181.

[7] GRAHAM, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics 17*, 2 (1969), 416–429.

[8] HEALY, C. A., WHALLEY, D. B., AND HARMON, M. G. Integrating the timing analysis of pipelining and instruction caching. In *RTSS'95* (Dec. 1995), pp. 288–297.

[9] KIRNER, R., KADLEC, A., AND PUSCHNER, P. Precise worst-case execution time analysis for processors with timing anomalies. In *ECRTS'09* (July 2009).

[10] LI, X., ROYCHOUDHURY, A., AND MITRA, T. Modeling out-of-order processors for WCET analysis. *Real-Time Systems 34*, 3 (November 2006), 195–227.

[11] LUNDQVIST, T. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Sweden, June 2002.

[12] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *RTSS'99* (Washington, DC, USA, 1999), IEEE Computer Society.

[13] PRATT, V. R. Two easy theories whose combination is hard. Tech. rep., Massachusetts Institute of Technology, 1977.

[14] REINEKE, J. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, November 2008.

[15] REINEKE, J., WACHTER, B., THESING, S., WILHELM, R., POLIAN, I., EISINGER, J., AND BECKER, B. A definition and classification of timing anomalies. In *WCET'06* (July 2006), Schloss Dagstuhl, Germany.

[16] ROCHANGE, C., AND SAINRAT, P. A context-parameterized model for static analysis of execution times. *Trans. on HiPEAC 2*, 3 (2007), 109–128.

[17] SCHNEIDER, J. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, Germany, Saarbrücken, Germany, December 2002.

[18] TARJAN, R. E. Shortest paths. Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[19] THEILING, H., FERDINAND, C., AND WILHELM, R. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems 18*, 2/3 (May 2000), 157–179.

[20] THESING, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[21] WENZEL, I., KIRNER, R., PUSCHNER, P., AND RIEDER, B. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference on Quality Software* (Sep. 2005).

STATISTICAL-BASED WCET ESTIMATION AND VALIDATION

Jeffery Hansen[1] Scott Hissam[1]
Gabriel A. Moreno[1]

Abstract
In this paper we present a measurement-based approach that produces both a WCET (Worst Case Execution Time) estimate, and a prediction of the probability that a future execution time will exceed our estimate. Our statistical-based approach uses extreme value theory to build a model of the tail behavior of the measured execution time value. We validate our approach using an industrial data set comprised of over 150 sampled components and nearly 200 million sample execution times. Each trace is divided into two segments, with one used to make the WCET estimate, and the second used check our prediction of the fraction of future execution time samples that exceed our WCET estimate. We show that compared to WCET estimates derived from the worst-case observed time, our WCET estimates significantly improve the ability to predict the probability that our WCET estimate is exceeded.

1. Introduction

In the analysis of many systems, it is necessary to know the Worst-Case Execution Time (WCET) of the tasks in the system. Traditionally, WCET estimation has been based on static-analysis techniques. The source code or disassembled binary executable of the task is analyzed to determine the time required for the longest path through the code. Modern processor techniques such as caching and predictive branching make this approach very difficult[3]. Efforts to overcome such difficulty require modeling and simulation of complex hardware architectures to augment these static-analysis code techniques[12, 7]. As a result, measurement-based approaches have become more popular.

In measurement-based approaches, estimates of the WCET of a task are made by running a series of sample executions of the task in isolation and directly measuring its execution time. This can be either the execution time of the whole task, or the execution time of individual basic blocks (segments of straight-line code with no intervening branches) of the task. In the simplest approaches, the task is executed and measured for a set number of times and the longest observed time, possibly scaled by an ad hoc “safety factor”, is used as the WCET. While this approach is simple, there is no systematic way to determine the appropriate scale factor or predict how good the WCET estimate will be.

Some approaches use a combination of analytical and measurement-based methods. For example a tool called pWCET [14, 13] breaks a piece of code down into basic blocks, uses measurment to compute an execution time distribution, then uses a set of techniques to compute joint distributions based on the type of control structures in the code. The authors describe a technique which can produce

[1]Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
{jhansen,shissam,gmoreno}@sei.cmu.edu

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2291

WCET predictions with no independence assumptions on the execution times of the individual basic blocks. The primary weakness of this approach is that the execution time is modeled directly by an empirical distribution function. Since WCET estimation is concerned primarily with the tail behavior of the distribution, a very large number of samples are required to obtain an accurate model.

While ideally we would like to find a WCET that is guaranteed never to be exceeded, this is not practical with a measurement-based approach. Instead the goal is to find WCET estimates that have very low and predictable probabilities of being exceeded (e.g., 10^{-5}).

1.1. Extreme Value Theory

In this paper, we present and evaluate a WCET estimation algorithm based on Extreme Value Theory (EVT). Extreme Value Theory[5, 1] is a branch of statistics for analyzing the tail behavior of a distribution. It allows us to reason about rare events, even though we may not have enough sample data to actually observe those rare events. Example applications include hydrology, finance and insurance.

1.2. WCET Estimation Using EVT

By using EVT, we can estimate WCET values that acheive a target exceedence probability (the probability that a future sample will exceed our WCET estimate). This is in contrast with the more primitive approach of simply using the highest observed execution time of a task over a set of trials for which we can not predict how well it will do in future invocations[10].

In one recent approach to using EVT for WCET estimation[2], execution time measurements are first fit to the Gumbel distribution[5] using an unbaised estimator. A WCET estimate is then made using a pertaining excess distribution function. The problem with this approach is that it incorrectly fits raw execution time data to the Gumbel distribution. The Gumbel and other EVT distributions are intended to model random variables that are the maximum (or minumum) of a large number of other random variables. In general, this is not the case for execution time measurements. An additional problem is that there does not appear to be any goodness-of-fit test to ensure the estimated parameters actually fit the measured data.

Our WCET estimation algorithm is also based on EVT. The most important change is that rather than attempting to fit the Gumbel distribution directly to the sample execution times, we use the method of block maxima[1], grouping the sample execution times into blocks and fitting the Gumbel only to the maximum values from each of the blocks. This ensures that the samples used to estimate the Gumbel distribution are samples from a maximum of random variables. In addition, we use a Chi-squared test to ensure that the sample data correctly matches the distribution we fit.

2. About the Data

In order to validate our WCET estimation approach, we use execution time traces from a set of 154 periodic tasks incorporating over 200 million combined execution time samples. These tasks make up a commercial device implemented as a PowerPC based embedded system running the VxWorks real-time operating system with many built-in monitoring and control interfaces. Two different instances of this device were stressed and measured by two different teams in two different laboratories using the same measurement procedure and device configuration over the course of a few days to obtain the observed execution times of these tasks[4]. Approximately 125 minutes of trace data was collected

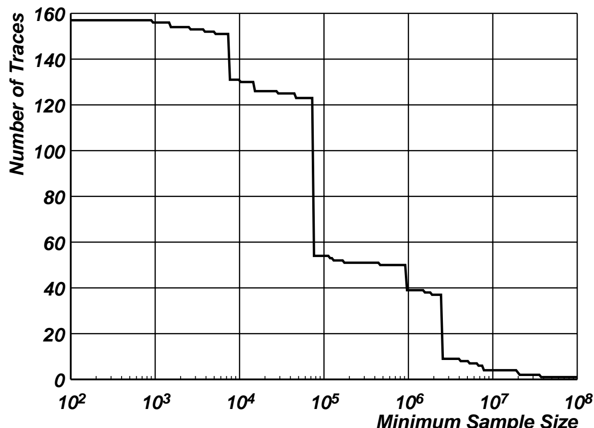


Figure 1. Number of Traces Meeting a Minimum Sample Size

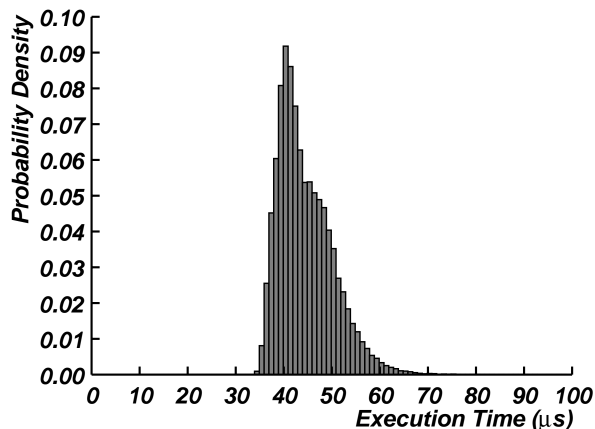


Figure 2. Execution Time Distribution for Task T_1

Property	Value
Trace Length	15 min.
Number of Samples	300,793
Execution Time Maximum	113.8 μs
Execution Time Mean	43.68 μs
Execution Time Std.	6.08 μs

Table 1. Execution Time Summary for Task T_1 Estimation Trace Data

for each task in 25 five-minute runs. Each trace is divided into two parts: an estimation part and a validation part. The estimation part is used for making WCET estimates and consists of the first 15 minutes of the trace data. The validation part is used for evaluating our WCET estimates and consists of the final 110 minutes of each trace. We used this partitioning between the estimation and validation parts because it gave us a good trade off between traces for which the WCET estimate converged while keeping the amount of estimation data low.

Since the periods of the tasks are different, the number of sample execution times collected in each trace is different. In order to ensure a sufficient number of samples for estimation and validation, we exclude those traces with less than 75,000 samples. We chose this value because it represents a large break in the number of traces exceeding this threshold (see Figure 1), and empirical evidence suggest that the 9,000 samples in the estimation part of the trace are usually sufficient to make a good estimate. A total of 122 tasks met this threshold.

2.1. Trace Characteristics

There are no data dependent loops in any of the tasks we measured. Because of this, we expect the execution times to be dependent only on factors such as the state of the cache, predictive branching and other processor effects. To generalize our results to tasks with data-dependent loops it would be necessary to combine our approach, (using it to estimate the execution time of basic blocks) with code analysis.

Table 1 lists basic statistics of the 15 minute estimation part of a representative trace which we will call T_1. The execution time distribution for this trace is shown in Figure 2. As can be seen from the graph, the distribution peaks near the mean and falls of with rapidly decreasing probability density

above 60μs.

2.2. Collection Methodology

Execution time measurements were collected from an embedded system (excluding the operating system itself) with over one million source lines of C and C++ code. Each task in the system, when running in steady state, was performed by an identifiable task body in the source code. Each task body was tagged by inserting WindView[11] user event markers in the source to identify when a task started working on a job and when that job was complete. As such, these markers delineated in the observed trace when a task was actually running in the CPU and when that task body had yielded. With these markers, the actual periods and execution times for each job the task performed could be measured.

Once the source code was tagged, the measurement procedure was to bring the device into steady state (post-startup phase) and collect measurement samples from the device using the same industrial testing methods and standards adopted by the device's manufacturer. This meant that the device was subjected to the maximum allowable data rates on all input channels for a fixed interval of time. After the sample data was collected, the traces from the jobs performed by the tasks were analyzed to calculate each jobs' execution time (the time occurring between each user event marker taking into account any preemption or blocking that may have occurred during the jobs execution). A more detailed discussion of the measurement approach can be found in [4].

3. Extreme Value Theory

In principle, we could collect enough sample execution time data to construct an empirical distribution function $\hat{F}(x)$ and then simply use the inverse CDF to compute an estimated WCET as $\hat{\omega} = \hat{F}^{-1}(1 - p_e)$ where p_e is the desired probability of exceeding our estimate. The problem with this approach is that in order to have a good model of the right tail behavior we would need a vast amount of data. Furthermore, we could never generate an estimate higher than the highest observed value. Extreme Value Theory (EVT)[5] gives us a way to reason about this tail behavior without the vast amount of data required by a brute-force approach.

Given a random variable $Y = \max X_1, ..., X_n$ formed from the maximum of a set of n i.i.d. (independent identically distributed) random variables X_i, EVT tells us the distribution of Y will converge to the Generalized Extreme Value (GEV) distribution as n goes to infinity. This is analogous to the role the central limit theorem plays in the sum of a large number of random variables. Properties of the distribution of the base random variables X_i determine to which of three forms the distribution of Y will converge. The three forms are: Type I (Gumbel) – When the underlying distribution has a non-heavy upper tail (e.g., Normal), Type II (Frechet) – When the underlying distributions has a heavy upper tail (e.g., Pareto), and Type III (Weibull) – When the underlying distributions has a bounded upper tail (e.g., Uniform).

In applying EVT to the WCET estimation problem, we assume that the execution time distribution (e.g., Figure 2) has a non-heavy tail for most tasks. This implies the GEV distribution will converge to the Type I or Gumbel form which we assume for the reminder of this paper. We test and confirm this hypothesis in Section 5..

The Gumbel distribution has two parameters: a location parameter μ and a scale parameter β. It has

1. *Set the initial block size* b *to 100.*
2. *If the number of blocks* $\lfloor N/b \rfloor$ *is less than 30, then stop (not enough samples to generate an estimate).*
3. *Segment execution times* $x_1, ..., x_N$ *into blocks of* b *measurements.*
4. *For each of the* $\lfloor N/b \rfloor$ *blocks find the maximum values* $y_1, ...y_{\lfloor N/b \rfloor}$ *where* $y_i = \max(x_{(i-1)b+1}, x_{(i-1)b+2}, ..., x_{ib})$.
5. *Estimate the best-fit Gumbel parameters* μ *and* β *to the block maximum values* $y_1, ...y_{\lfloor N/b \rfloor}$.
6. *If the Chi Squared fit between the block maximum values* $y_1, ...y_{\lfloor N/b \rfloor}$ *and the Gumbel parameters* μ *and* β *does not exceed a 0.05 confidence value, then double the value of* b *and go back to Step 2.*
7. *Return WCET estimation* $\omega = F_G^{-1}((1-p_e)^b)$ *where* F_G^{-1} *is the percent-point function (Equation 1) for a Gumbel with parameters* μ *and* β*,* b *is the block size and* p_e *is target sample exceedance probability.*

Figure 3. WCET Estimation Algorithm

the cumulative distribution function (CDF) $F_G(y) = e^{-e^{-\frac{y-\mu}{\beta}}}$ and the percent-point function:

$$F_G^{-1}(q) = \mu - \beta \log(-\log(q)) \quad (1)$$

4. WCET Estimation

In order to apply these EVT techniques to execution time samples, we must construct sample data from a random variable that is formed from the maximum of another set of random variables. We group the execution time samples into size b blocks of consecutive samples, then choose the maximum value from each block to construct a new set of sample "block maximum" values. That is, given a set of raw execution time measurements $x_1, ..., x_N$, we construct the set of block maximums $y_1, ..., y_{\lfloor N/b \rfloor}$ where y_i is the maximum x_j value between $x_{(i-1)b+1}$ and x_{ib}. Left over samples not filling a block are discarded.

By selecting only the block maximum values, the algorithm presented here focuses on the samples of most interest to us. In general the larger the block size b, the better fit we will get to the Gumbel distribution. However, there is a tradeoff since a increasing b will result in fewer blocks and thus fewer sample values.

Our algorithm for making WCET estimates takes as input a set of N execution time samples, $x_1, ..., x_N$, and a target exceedance probability p_e, and returns a WCET estimate for which we predict that future invocations of that task will exceed this estimate with probability p_e. The smaller the value we choose for p_e, the larger the estimated WCET value will be. Since the underlying algorithm assumes that execution times come from an unbounded distribution (but with rapidly decreasing probability density for large values), we can not generate a WCET value for $p_e = 0$.

Figure 3 shows an outline of our WCET estimation algorithm. This algorithm is discussed in greater

detail in the following sections. We will use the 15 minute estimation part of the trace for Task T_1 mentioned in Section 2.1. as an example.

4.1. Steps 1-4: Blocking of Samples

The goal of Steps 1 through 4 is to select a subset of the original data points that represent the upper tail of the behavior. This is done by grouping the data into blocks of b samples and discarding all but the maximum value from each block. Samples at the end of the trace that do not completely fill a block are discarded. We make the assumption that all execution time samples are independent and thus the blocking method chosen will not affect the statistical properties of the blocks.

The selection of b is a trade-off between the quality of fit to the Gumbel distribution, and the number of samples that can be used in making that fit. Generally, the larger the value we choose for b, the more likely the block maximum values will follow a Gumbel distribution, but the fewer samples we will have available to use in the estimation of the Gumbel parameters. For example, if we have 50,000 execution time samples with which to make an estimate, there will be 500 block maximum values when the block size is 100. But if we use a block size of 1,000, then there will be only 50 block maximum values to use in the estimation of the Gumbel parameters.

We use the generally accepted value of 30 as the minimum number of samples (block maximum values) to use in estimating a distribution. This limit also bounds the size to which b can grow. Since b is doubled until the goodness-of-fit test is passed (at Step 6), it is possible that $\lfloor N/b \rfloor$ can fall below 30 and thus we must stop the algorithm without generating a WCET estimate. In this case, the only remedy is to either collect more execution time samples or use another WCET estimation technique.

4.2. Step 5: Parameter Estimation

In Step 5, we compute the parameters of the Gumbel distribution of the block maximum values. We do this by applying linear regression to the QQ-plot of these values. A QQ-plot[1], or quantile plot, is a plot of the empirical quantile values of sample data against the quantiles of the standard form of a target distribution. For Gumbel quantiles, this is done by plotting the points:

$$(-\log(-\log(1 - \frac{i}{\lfloor N/b + 1 \rfloor})), y_i^{[s]}), \quad i = 1..\lfloor N/b \rfloor$$

where $\{y_1^{[s]}, ..., y_{\lfloor N/b \rfloor}^{[s]}\}$ are the block maximum values sorted from lowest to highest. If the block maximum values follow a Gumbel distribution, then the points on the QQ-plot will form a straight or nearly straight line. The slope and intercept of the best-fit line through these points can be used as estimators for the Gumbel μ and β parameters, respectively. While using linear regression to compute a best fit line is not the only method to estimate the Gumbel parameters, it is quicker and easier than methods such as maximum likelihood estimation.

An example QQ-plot for Task T_1 is shown in Figure 4. The dots represent the block maximum values, and the line represents the Gumbel fit to the data. Notice that we get a good linear fit to the data and we can find the Gumbel location parameter value $\mu = 61.72$ from the y-intercept, and the Gumbel scale parameter value $\beta = 5.95$ from the slope.

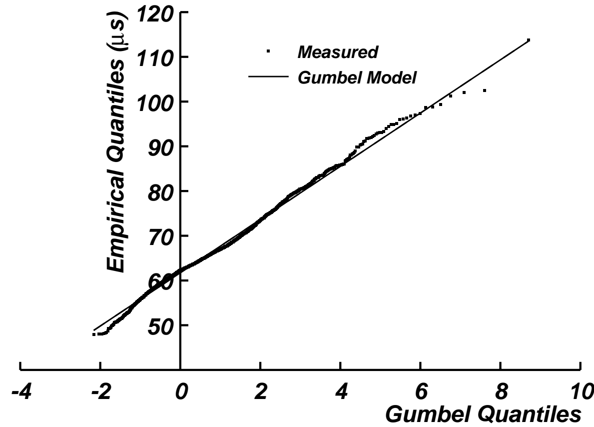


Figure 4. QQ-Plot of Block Maximum Values with Gumbel Quantiles for Task T_1

4.3. Step 6: Verifying Goodness-of-Fit

In Step 6, we verify the goodness of fit between the sampled block maximum values and the estimated Gumbel parameters using a Chi Squared test. The test is performed by grouping the sample values into M bins and then comparing the number of samples that fall into each bin O_i with the expected number of samples that fall into bin E_i (computed by multiplying the total number of samples by the difference in the Gumbel CDF at the edges of the bin) using the Chi Squared formula:

$$\Xi^2 = \sum_{i=1}^{M} \frac{(O_i - E_i)^2}{E_i} \tag{2}$$

The resulting value, along with the number of degrees of freedom is then checked against a Chi Squared Table to see if it is significant. The number of degrees of freedom in this case is given by $M - 3$. This includes two for the number of Gumbel parameters we are estimating, and one for the last bin who's value is fixed after all other bin values have been determined. From the Chi Square table we can find the critical Chi Square value for a level of significance p. p represents the probability that a Chi Square distributed random variable will exceed that critical value. Typically a match at the $p = 0.05$ is considered acceptable.

In our algorithm, we create the bins by dividing the range between the smallest and largest block maximum value into $N_b/30$ equally spaced bins, where $N_b = \lfloor N/b \rfloor$ is the number of blocks. This will create an initial set of bins for which the average number of observations falling into them is 30. However, we do not let the number of bins fall below six. Since the Chi squared test is sensitive to bins with low numbers of observations, we collapse adjacent bins with less than five observations. This is done by scanning the blocks from the lowest to the highest combining blocks with less than five observations in it until there are five or more observations. Again, we always retain at least six bins.

For the example task T_1 at a block size of $b = 100$ there were 3007 block maximum values which resulted in an initial set of 100 bins for the Chi Squared test. Figure 5 shows the number of block maximum values that fell in each of the bins (bars) along with the expected values for each of these bars (the solid line). After collapsing small bins, we got a total of 62 bins and computed a Chi Square value of 236.0. Since this exceeds the required Chi-Squared value of 77.93 for $59 = 62 - 3$ degrees

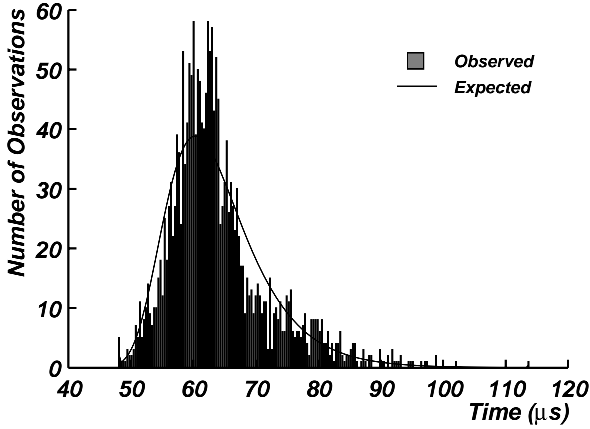


Figure 5. Fit of Task T_1 Block Maximums Against Gumbel for $b = 100$

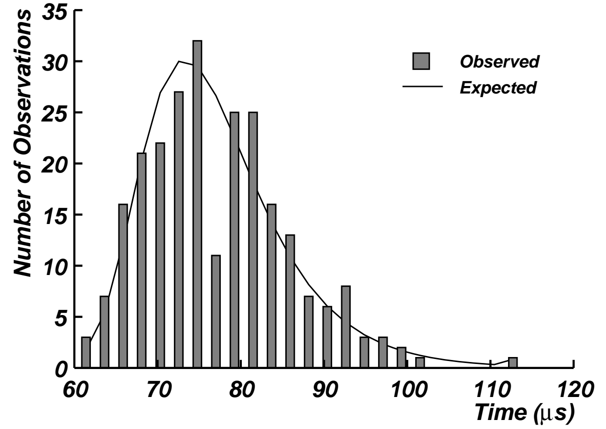


Figure 6. Fit of Task T_1 Block Maximums Against Gumbel for $b = 400$

of freedom at the $p = 0.05$ confidence value, we must reject the hypothesis that the observed data matches a Gumbel distribution.

Since the goodness-of-fit test failed, Step 6 calls for us to double the block size and try again. Doing this repeatedly, we find that at a block size of $b = 400$, we obtain a Chi Squared value that does not require us to reject a match between the estimated Gumbel parameters and the block maximum data. In this case we got Gumbel parameters of $\mu = 70.0$ and $\beta = 6.23$. Note that these values are not directly comparable with the $b = 100$ case since they are estimated from a different set of values.

In applying the goodness-of-fit test to the $b = 400$ case, we initially get 25 bins which collapsed into 19 bins after we combine the bins with fewer than five observations. This results in 16 degrees of freedom and thus a critical Chi squared value of 26.3 for a fit at the $p = 0.05$ confidence level. Figure 6 shows the fit between the observed bin counts (bars) and the expected bin counts (line) for this case.

4.4. Step 7: Estimating WCET Value

The final step is to use the computed and verified Gumbel parameters and the exceedance probability p_e to estimate the WCET. This is done using the Gumbel percent-point function (Equation 1). This function returns the block maximum value for which there is a probability q that a measured block maximum will not exceed this value. We can compute q as:

$$q = (1 - p_e)^b \tag{3}$$

That is, q is the probability that all b samples in the block are below the WCET value. Combining Equations 3 and 1 gives us the equation:

$$\omega = \mu - \beta \log(-\log((1 - p_e)^b)) \tag{4}$$

for computing the WCET estimate ω in terms of the block size b, the exceedance probability p_e, and the Gumbel parameters μ and β. In the example for Task T_1, at a target exceedance probability of $p_e = 10^{-4}$ we get the WCET estimate:

$$\omega = 70 - 6.23 \log\left(-\log\left((1 - 10^{-4})^{400}\right)\right) = 90.05\mu\text{s}$$

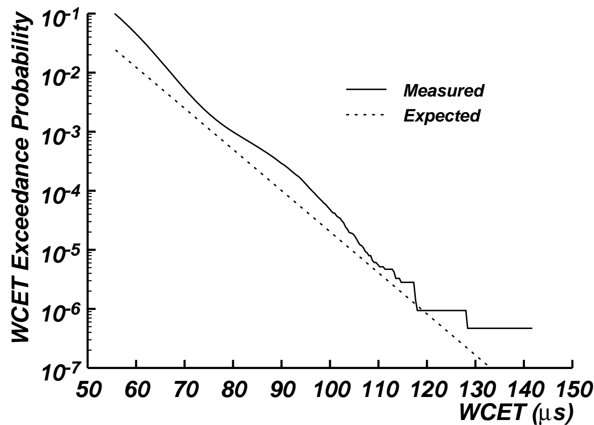


Figure 7. Predicted vs. Measured Exceedance Probability for Task T_1

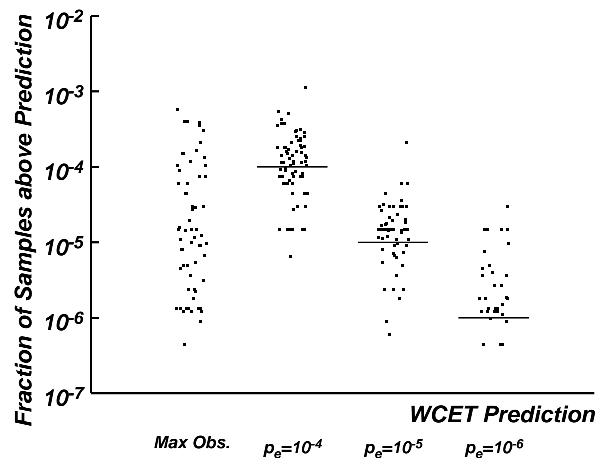


Figure 8. Distribution of Exceedance Probability for All Tasks

5. Validation

In applying our WCET estimation algorithm to the 15 minute estimation part of the 122 traces described in Section 2., 75 of the traces passed the Chi squared test and yielded a WCET estimate. These tasks will be the focus of our analysis of the quality of the WCET estimates. However, we point out that by increasing the amount of data used by the estimation algorithm from 15 minutes to 30 minutes, the number of traces for which we could make an estimate was increased to 95.

5.1. Single-Task Validation

A comparison between the predicted and the measured exceedance probability for Task T_1 example is shown in Figure 7. The predicted exceedance probability curve was generated from Equation 4 using the parameters estimated in the previous sections($\mu = 70.0$, $\beta = 6.23$, and $b = 400$). The measured curve was generated by sweeping through WCET values and measuring the number of samples that fall above each value. A total of about 2.2 million execution times where used to validate the WCET estimates for T_1.

We can see that the predicted exceedance probability matches well with the measured probability. In particular, the slopes (on a log scale) match extremely well. However, there is some divergence, particularly in the lower-right corner of the graph where the measured curve takes on a “stair step” appearance. This “stair step” effect is due to sample size limitations at the right-hand side of the graph. Even with 2.2 million samples, single values passing below the WCET estimate can cause large changes when measuring events near the 10^{-6} probability level.

We also examined the curves for the predicted versus observed exceedance probabilities for the other 74 tasks for which we generated a WCET estimate. We found results similar to those for task T_1. While a slight positive or negative bias between the observed and predicted exceedance probability curves was present in most traces, the slopes generally matched extremely well.

5.2. Task Set Validation

We compared the WCET estimates made using our WCET estimation algorithm with a “Maximum Observed” method in which the highest observed value in the estimation part of the trace was taken

as the WCET estimate.

Figure 8 shows a set of four point clouds showing the fraction of validation execution time samples (the last 110 minutes of each trace) that exceeded the WCET estimate for the Maximum Observed method and our WCET estimation method at three different p_e values. Each dot in each point cloud represents a single trace and its fraction of WCET exceedances. It can be seen that in the "Maximum Observed" case, the fraction of WCET exceedances is very unpredictable and has a high variance. In contrast, the variability for our EVT-based approach is significantly smaller and clustered around the predicted value indicated by the horizontal lines. Note that while the measured exceedance probabilities for the $p_e = 10^{-6}$ case appears to be biased toward exceedance probabilities higher than 10^{-6}, there are actually 42 tasks for which the measured exceedance probability was zero and thus can not be seen on the graph. This $p_e = 10^{-6}$ case is in fact near our limit to measure the WCET exceedances given the amount of validation data we have.

6. Conclusion

The most important aspect of a measurement-based WCET estimation method is the ability to predict the exceedance probability. By using a large collection of traces, we have shown that our EVT-based method can produce WCET estimates for which the exceedance probability is more predictable and controllable than using the maximum observed execution time.

References

[1] BEIRLANT, J., GOEGEBEUR, Y., SEGERS, J., and TEUGELS, J., "Statistics of Extremes: Theory and Applications," Wiley Press, 2004

[2] EDGAR, S., BURNS, A., "Statistical Analysis of WCET for Scheduling," in *Proceedings of the 22nd Real-Time Systems Symposium*, pg. 215-224, 2001

[3] HILLARY, N. and MADSEN, K., "You can't control what you Can't Measure, OR Why it's Close to Impossible to Guarantee Real-time Software Peformance on a CPU with on-chip cache," in *Proc. of the 2nd Int. Workshop on WCET Anal.*, June 2002

[4] HISSAM, S. A., MORENO, G. A., PLAKOSH, D., SAVO, I. and STELMARCZYK, M., "Predicting the Behavior of a Highly Configurable Component Based Real-Time System", in Proc. of 20th Euromicro Conf. on Real-Time Systems (ECRTS 08), 2008

[5] GUMBEL, E.J., "Statistics of Extremes", Columbia University Press, 1958

[6] LEAHY, K., "Efficient Estimation of Tighter Bounds for Worst Case Execution Time of Programs", Matser's Thesis, Dep. of Comp. Sci. and Eng., Washington Univ. in St. Louis, December 2004

[7] LIM, S.S., BEA, Y. H., JANG, G. T., RHEE, B.D., MIN, S.L., PARK, Y.C., SHIN, H. and KIM. C.S., "An accurate worst case timing analysis for RISC processors". In IEEE Real-Time Systems Symposium, pages 97–108, December 1994

[8] PERRONE, R., MACÊDO, R. and LIMA, G., "Estimation Execution Time Probability Distributions in Component-Based Real-Time Systems," in *Proceedings of the 10th Brazilian Workshop on Real-Time and Embedded Systems*, May 2008

[9] PETTERS, S., "How much Worst Case is Needed in WCET Estimation?", in *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis*, June 2002

[10] SCHAEFER, S., SCHOLZ, B., PETTERS, S. M. and HEISER, G., “Static Analysis Support for Measurement-based WCET Analysis”. In 12th IEEE Int. Conf. on Embed. and Real-Time Comp. Sys. and App., Work-in-Progress Session, August 2006

[11] WIND RIVER SYSTEMS, INC., “Tornado Getting Started Guide, 2.2, Windows Version”, Wind River Systems, Inc., Alameda, CA, 2002

[12] ZHANG, N., BURNS, A. and NICHOLSON, M., “Pipelined Processors and Worst Case Execution Times”, Real Time Systems, Vol. 5, pp. 319-343, 1993

[13] BERNAT, G., COLIN, A. and PETTERS, S., “pWCET: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems”, in *Proc. of the ACM 2003 Conf. on Languages, Compilers, and Tools for Embedded Sys. (LCTES)*, 2003

[14] BERNAT, G., COLIN, A. and PETTERS, S., “WCET Analysis of Probabilistic Hard Real-Time Systems”, in *Proc. of the 22nd IEEE Real-Time Sys. Symp.*, 2002

WCET ANALYSIS OF MULTI-LEVEL SET-ASSOCIATIVE DATA CACHES

Benjamin Lesage , Damien Hardy and Isabelle Puaut[1]

Abstract

Nowadays, the presence of cache hierarchies tends to be a common trend in processor architectures, even in hardware for real-time embedded systems. Caches are used to fill the gap between the processor and the main memory, reducing access times based on spatial and temporal locality properties of tasks. Cache hierarchies are going even further however at the price of increased complexity. In this paper, we present a safe static data cache analysis method for hierarchies of non-inclusive caches. Using this method, we show that considering the cache hierarchy in the context of data caches allows tighter estimates of the worst case execution time than when considering only the first cache level. We also present considerations about the update policy for data caches.

1. Introduction

It is crucial in hard real-time systems to prove that tasks meet their deadlines in all situations, including the worst-case. This proof needs an estimation of the worst-case execution time (WCET) of every task taken in isolation. WCET estimates have to be safe, i.e. larger than or equal to any possible execution time. Moreover, they have to be tight, i.e. as close as possible to the actual WCET. Thereof, WCET estimation techniques have to account for all possible execution paths in the program and determine the longest one (*high-level* analysis). They also have to account for the hardware the task is running on (*low-level* analysis).

Cache memories are introduced to decrease the access time to the information due to the increasing gap between fast micro-processors and relatively slower main memories. Architectures with caches are now commonly used in embedded real-time systems due to the increasing demand for computing power of many embedded applications. The presence of caches in real-time systems makes WCET estimation difficult due to the dynamic behaviour of caches. Safely estimating WCET on architectures with caches requires a knowledge of all possible cache contents at every program point, and requires some knowledge of the cache replacement policy and, in case of data caches, update policy.

During the last decade, much research has been undertaken to predict WCET in architectures equipped with caches. Regarding instruction caches, static cache analysis methods [13, 14, 18, 3] have been designed and recently, extended to hierarchies of non-inclusive caches [7]. To overcome predictability issues, as the ones due to the replacement policies, is the family of approaches like locking [15, 20]. The latter methods family suits well to data caches [19, 11], whose analysis suffers from imprecise static accessed data address prediction. Indeed, if precise address prediction in the context of instruction caches has been mastered, for data caches, it remains an important concern. Nonetheless, existing methods for instruction caches have also been modified [16, 17, 4] to tackle with accesses whose target can only be over-approximated using a range of addresses.

To the best of our knowledge, no safe static cache analysis method has been proposed so far to predict worst-case data cache behaviour in the presence of a data caches *hierarchy*. The issues to be tackled when designing such an analysis are twofold. On the one hand, the prediction of cache levels impacted

[1]IRISA, University of Rennes 1, Rennes, France

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2283

by a memory reference is required to estimate the induced caches accesses. On the other hand, writes to the caches have to be considered, because they may introduce additional cache accesses. Ensued predictability problems may even get exacerbated by the lack of precise knowledge about accessed addresses.

The contribution of this paper is the proposal of a new safe static cache analysis method for multi-level non-inclusive set-associative data caches. All levels of cache are analysed sequentially. Similarly to our previous work for instruction caches [7], the safety of the proposed method relies on the introduced concept of a *cache access classification*, defining which references may occur at every cache level and have to be considered by the cache analysis of that level, in conjunction with the more traditional *cache hit/miss classification*. This paper presents experimental results showing that in most cases WCET estimates are tighter when considering the cache hierarchy than when considering the L1 cache only.

The rest of the paper is organized as follows. Related work is surveyed in Section 2. Section 3 presents the type of caches to which our analysis applies. Section 4 then details our proposal. Experimental results are given in Section 5. Finally, Section 6 concludes with a summary of the contributions of this paper, and gives directions for future work.

2. Related work

Caches in real-time systems raise timing predictability issues due to their dynamic behaviour and their replacement policy. Many static analysis methods have been proposed in order to produce a safe WCET estimate on architectures with caches. To be safe, existing cache analysis methods determine *every* possible cache contents at every point in the execution, considering all execution paths altogether. Possible cache contents can be represented as sets of *concrete cache states* [9] or by a more compact representation called *abstract cache states* (ACS) [18, 3, 14, 13].

Two main classes of approaches [18, 13] exist for the static WCET analysis on architectures with a single level of instruction cache. In [18] the approach is based on *abstract interpretation* [2] and uses ACSs. In this approach, three different analyses are applied which use fixpoint computation to determine if a memory block is *always* present in the cache (*Must* analysis), if a memory block *may* be present in the cache (*May* analysis), or if a memory block will not be evicted after it has been first loaded (*Persistence* analysis). A *cache hit/miss classification* (e.g. *always hit*, *first miss*...) can then be assigned to every instruction based on the result of the three analyses. This approach originally designed for set-associative instruction caches implementing the *least recently used* (LRU) replacement policy has been extended for different cache replacement policies in [8] for instruction caches. In [13], *static cache simulation* is used to determine every possible content of the instruction cache before each instruction. Static cache simulation computes abstract cache states using data-flow analysis. A *cache hit/miss classification* is used to classify the worst-case behaviour of the cache for a given instruction. The base approach, initially designed for direct-mapped caches, was later extended to set-associative instruction caches in [14].

A peculiarity of data caches, compared to instruction caches, arises as the precise target of some references may not be statically computable. A first solution is to consider these imprecise accesses as in [17] and [4], improvements to [18]. Therefore, we base our cache analysis on these studies.

An alternative solution to deal with data caches are *Cache Miss Equations* (CME) [19, 11, 21]. To estimate cache behaviour, the iteration space of loop nests is represented as a polyhedron. *Reuse vectors* [22] are then defined between points of the iteration space. CME are set up and resolved to accurately locate misses. This method has been successfully applied to data caches in combination

with locking [19]. However, according to the authors, this approach suffers from a lack of support for data dependent conditionals.

Another scarcely addressed characteristic of data caches is the impact of memory modifying instructions. In [5], an analysis was proposed based on the *write-back* update policy. Modified data in a cache are copied back to the main memory upon eviction. The objective is thus to estimate the *write backs*, i.e. the moment when a modified data might be replaced in the cache. In a first step towards a more generic solution, we chose to use the *write-through* update policy as it removes the need of *write backs* monitoring.

Finally, about the hierarchy of data caches, we already explored a solution to this problem in [7] in the context of instruction caches. Introducing the concept of *cache access classification* (CAC), we safely identify references that may, must or never occur at every level in the cache hierarchy. This paper presents to which extent this approach can be applied to data caches.

3. Assumptions and notations

As this study focuses on data caches, code is assumed not to interfere with data in the different considered caches. There is no assumption on the means used to achieve this separation, whether they are software or hardware based. An architecture without timing anomalies, caused by interactions between caches and pipelines [12], is however assumed.

The considered cache hierarchy is composed of N levels of data caches. Each cache implements the LRU replacement policy. Using this policy, cache blocks in a cache set are logically ordered according to their age. If a cache set is full, upon a load in this set, the evicted block is the oldest one whereas the most recently accessed block is the youngest one.

A datum accessed by an instruction should be located in a single memory block. Cache line size of level L is assumed to be a multiple of the cache line size of level $L-1$. However, no assumption is made concerning the cache sizes or associativities. Furthermore, the following properties are assumed to hold:

P1.[**load**] A piece of information is searched for in the cache of level L if, and only if, a cache miss occurred when searching it in the cache of level $L-1$. Cache of level 1 is always accessed.
P2.[**load**] Every time a cache miss occurs at cache level L, the entire cache line containing the missing piece of information is loaded into the cache of level L.
P3.[**store**] The modification issued by a store instruction goes all the way through the memory hierarchy. Writes to the cache levels where the written memory block is already present are triggered, along with the update of the main memory. Otherwise, if the information is absent from a cache, this cache level is left unchanged.
P4.[**store**] Upon a write in a cache block, wherever is the cache in the hierarchy, no block age modification is induced[2].
P5. There are no action on the cache contents (i.e. lookup/modification) other than the ones mentioned above.

Property P1 rules out architectures where cache levels are accessed in parallel to speed up information lookup. P2 excludes architectures with exclusive caches, whereas P5 filters out cache hierarchies ensuring inclusion.

Properties P3 and P4 on the other hand address the store instructions behaviour. P3 ensures the use of

[2]Note that this is not a strong assumption but its alleviation is left as future work.

the *write-through* update policy. In combination with P3, P4 corresponds among other things to the *write-no-allocate* update policy for members of the cache hierarchy, as illustrated in Figure 1.

Finally, the latencies to access the different levels of the memory hierarchy are assumed to be bounded and known.

We define a *memory reference* as a reference to *data* in the memory triggered by a load or store instruction in a fixed call context; a memory reference is tied to a unique instruction and a unique call context.

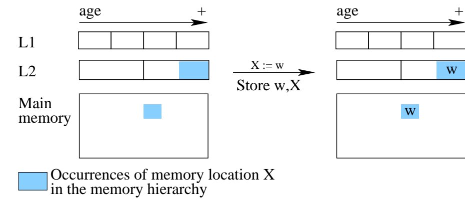


Figure 1: Example of the memory hierarchy behaviour upon a store instruction (write-through and write-no-allocate policies).

4. Data Cache Analysis

This section introduces most of the involved steps in our computation of WCET contribution for hierarchies of data caches. The structure of the whole method is outlined in Figure 2.

After a first step that extracts a *Control Flow Graph* from the analysed executable, a data address analysis (§ 4.1) is performed. The objective is to attach to every memory reference a safe estimate of the accessed addresses.

Then, the caches of the hierarchy are analysed one after the other (§ 4.2) based on address information. For each cache level and each instruction issuing memory operations, a *cache hit/miss classification* (CHMC) is computed. These classifications represent the worst-case behaviour of this cache level with regards to this instruction.

To be safe, the analysis of a cache level further relies on *cache access classifications* (CAC, introduced in § 4.3) which discloses, given a memory reference, a safe approximation of whether or not it occurs at a cache level.

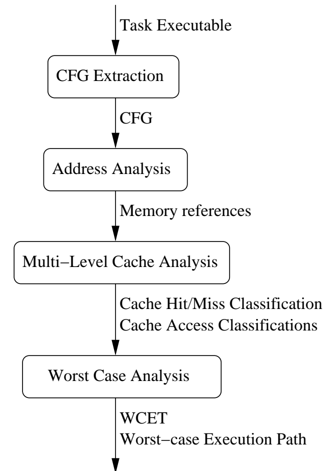


Figure 2: Complete task analysis overview

In the end, a timing analysis of memory references (§ 4.4), considering data caches, is performed with the help of both the CHMC and CAC for each cache level. Such information may then be used to compute the longest possible execution path and finally the WCET of the task.

4.1. Address analysis

The address analysis, in the context of data caches, computes for every memory reference, the memory location it may access. Such information may however not be precisely computable, hence producing an over-approximation to yield safe values for subsequent analyses. These over-approximations take the form of ranges of possibly accessed memory blocks instead of a reference to a precise memory block.

The address analysis uses data-flow analyses which first computes stack frames addresses for each valid call context and then analyse register contents for each basic block. Considering both global and on stack accesses, the precise address of a scalar is yielded whereas the whole array address range is returned for accesses to array elements. Note that the analysis used below is the one proposed in [6].

4.2. Single level data cache analysis

Below, a data cache analysis abstracted from the multi-level aspect is introduced. All references are analysed and access filtering, according to other cache levels, is introduced later (§ 4.3). The analysis method is presented along with the different mechanisms to handle the specificities of data caches, compared to instruction caches.

Focusing on data caches, the timing analysis of memory references is performed using the worst-case behaviour of this data cache for each memory reference. This knowledge is itself based on the cache contents at the studied program point, thus requiring a safe estimate of memory blocks present in the cache.

A *cache hit/miss classification* (CHMC) is used to model the defined cache behaviours with regards to a given memory reference. To such a reference has already been attached a set of possibly accessed memory blocks by the address analysis. This set of memory blocks is used to compute the CHMC:

always-hit (AH) : all the possibly accessed memory blocks are guaranteed to be in the cache;
first-miss (FM) : for every possibly accessed memory block, once it has been first loaded in the cache, it is guaranteed to stay in the cache afterwards;
always-miss (AM) : all the possibly accessed memory blocks are known to be absent from the cache;
content-independent (CI) : if the behaviour of the memory reference does not depend on the cache contents. The CI classification is used for store instructions, which, according to our hypotheses (§ 3) does not depend on the cache contents;
not-classified (NC) : if none of the above applies.

Abstract cache states (ACSs) are used to collect information along the task CFG. This abstraction allows the modelling of a combination of concrete cache states, in terms of present memory blocks and their relative age. This is required as all paths have to be considered altogether to yield safe values. Different analyses, similarly to [18], are defined to collect information about cache contents at every program point. For each analysis, fixpoint computation is applied on the program CFG, for every call context:

a *Must* analysis determines if a memory block is always present in the cache, at a given point, thus allowing a *always-hit* classification;
a *Persistence* analysis determines if a memory block will not be evicted from the cache once it has been first loaded, as in the definition of the *first-miss* classification;
a *May* analysis determines if a memory block may be in the cache at a given point, otherwise the block is guaranteed not to be in the cache thus possibly allowing a *always-miss* classification. If, at a given point, some possibly accessed memory blocks are present in the May analysis but neither in the Must nor the Persistence one, the *not-classified* classification is assumed for this memory reference.

Then, for each memory reference, its set of possibly accessed memory block is compared to the memory blocks inside the input ACS computed by each analysis.

Join functions : For instructions on branch reconvergence, the $Join_{Must}$, $Join_{Persistence}$ and $Join_{May}$ (respectively for the Must, Persistence and May analysis) are used as a mean to compute input ACS:

$Join_{Must}$ computes the intersection of memory blocks present in the input ACSs, keeping for each one its maximal age as shown in Figure 3a;
$Join_{Persistence}$ keeps the union of memory blocks present in the input ACSs, as for $Join_{Must}$, the maximal age of memory blocks is kept;

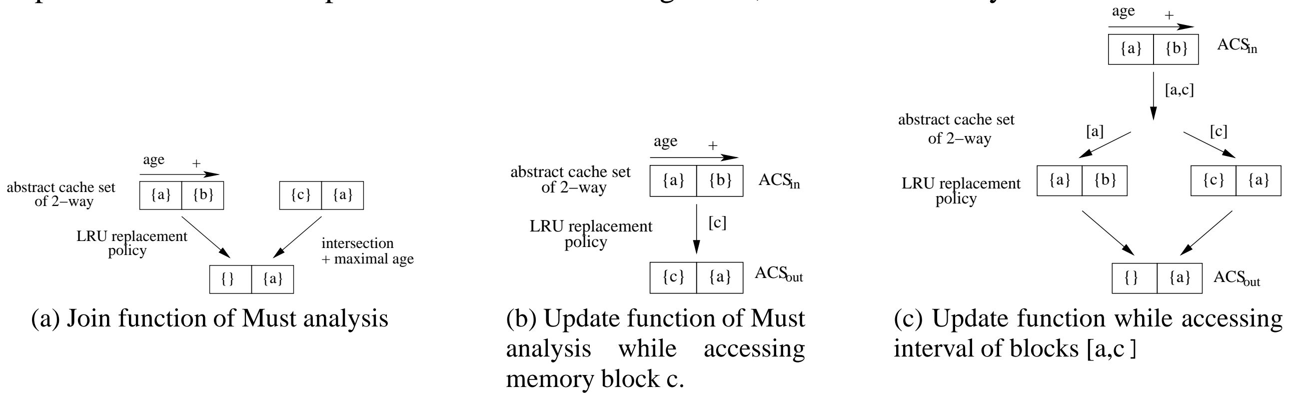
Join$_{May}$ computes the union of memory blocks present in the input ACSs, keeping for each one its minimal age.

Considering data caches, these functions are the same as the ones defined for instruction caches [18].

Update functions : The effects of memory references on the cache are modelled using the *Update*$_{Must}$, *Update*$_{Persistence}$ and *Update*$_{May}$ for the Must, Persistence and May analysis respectively. In all the cases, only load instructions have an impact on the cache contents. Store instructions have no impact on ACSs (§ 3, use of the write-through and write-no-allocate update policies). Considering data caches, the *Update* function of the different analyses is of particular interest. Indeed, it has to deal with accesses indeterminism, when a precise memory location cannot be statically defined for a load instruction. Two options have to be considered.

On the one hand, the precise accessed memory block may have been computed by the address analysis. The *Update* function is then pretty straightforward, as illustrated in Figure 3b for the Must analysis. Thus considering the *Update*$_{Must}$, the accessed block is put at the head of its cache set and the younger lines are shifted, i.e. made older and evicted if too old. Note that memory blocks outside an ACS are supposed to be older than the ones present in this same ACS. Furthermore, the different *Update* functions behaviour in this case is the same than for instructions [18].

On the other hand, when the address analysis yields a set of possibly accessed memory blocks for a memory reference, only one member of this set is actually accessed. To model this behaviour, a copy of the input ACS is created, using the appropriate *Update* function, for each possibly accessed memory block. Then, all the updated copies are unified, this time using the appropriate *Join* function to produce an ACS. This process is illustrated on Figure 3c, for the Must analysis.



(a) Join function of Must analysis

(b) Update function of Must analysis while accessing memory block c.

(c) Update function while accessing interval of blocks [a,c]

Figure 3: *Update*$_{Must}$ and *Join*$_{Must}$ functions

Termination of the analyses. In the context of abstract interpretation, to prove the termination of an analysis, it is sufficient to prove the use of a finite abstract domain and the monotony of the transfer functions. ACS domain was shown to be finite in [18]. Moreover, [18] demonstrates the monotony of the *Join*$_x$ and *Update*$_x$ functions ($x \in \{Must, Persistence$ or $May\}$), for instructions and thus for accesses to precise memory blocks. In our case, the modification to be proved is the one applied to this same *Update*$_x$ function to handle accesses to possibly referenced memory blocks.

When an ACS is updated using a set of possibly referenced memory blocks for an analysis x, a composition of the *Update*$_x$, for each block, and the *Join*$_x$ function is performed. As the composition of monotonic functions is monotonic, our modifications ensure monotony. □

4.3. Multi-level analysis

The cache analysis described in § 4.2 does not support hierarchies of caches, i.e. all references are considered for the caches by the analysis. But, a memory reference may not occur in all the

cache levels of the hierarchy. This filtering by previous caches in the hierarchy, impacts the cache contents. Hence, caches are analysed one after the other, from cache level 1 to N, and *cache access classifications* (CACs), as defined in [7], are used to represent occurrences of memory reference r on cache level L. This is possible as, according to our hypotheses, occurrences of memory references on cache level L (and thus L contents) only depend on occurrences of memory references on caches levels $L' < L$.

Different classifications have been defined to represent if memory reference r is performed on cache level L:

Always (A) means that the access *r* is always performed at cache level *L*,

Never (N) means that the access *r* is never performed at cache level *L*,

Uncertain-Never (U-N) indicates that no guarantee can be given considering the first access to each possibly referenced memory block for r, but next accesses are never performed at level L,

Uncertain (U) indicates that no guarantee can be given about the fact that the access to *r* will or will not be performed at level *L*.

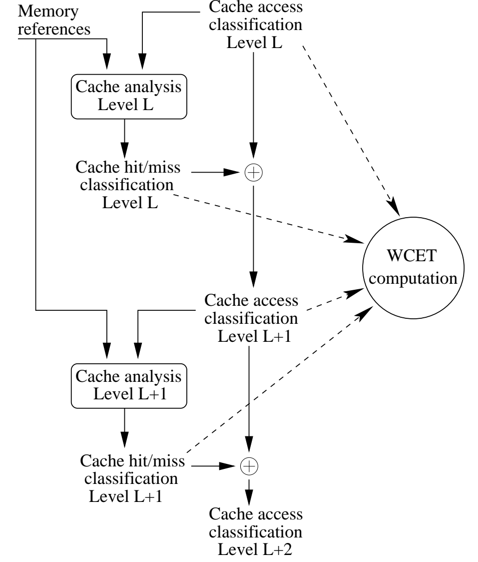


Figure 4: Multi-level non-inclusive data cache analysis framework

For the L1 cache, CAC determination is simple as it was earlier assumed that all memory references will be performed in this cache level (P1. § 3). This implies a A classification for every memory reference, considering the L1 cache.

Concerning greater cache levels, the CAC for memory reference r and cache level L is defined using both the CAC and the CHMC of the previous level cache level (see Figure 4) as in [7], to safely model cache filtering in the cache hierarchy. This is illustrated in Table 1.

Once these classifications have been settled, they have to be considered in the multi-level cache analysis. The modifications to handle hierarchies of data caches are similar to the modifications to handle hierarchies of instruction caches [7]. A (respectively N) classification for a given instruction, means that the reference is (respectively not) performed on the considered cache level which should be modelled by the analysis. As for the U and U-N classifications, both the cases are possible: the access is performed (A) or not (N) on this cache level. Similarly to non deterministic accesses, the two alternatives are considered and their outcomes unified using the appropriate *Join* function of the analysis.

$CAC_{r,L-1}$ \ $CHMC_{r,L-1}$	AH	FM	AM	NC
A	N	U-N	A	U
N	N	N	N	N
U-N	N	U-N	U-N	U-N
U	N	U-N	U	U

Table 1: Cache access classification for level L ($CAC_{r,L}$)

The *Update*$_x$ function, $x \in \{Must,\ Persistence$ or $May\}$, is modified to consider these classifications :

$$Update_x^{ml}(ACS_{in}, r, L) = \begin{cases} Update_x(ACS_{in}, r, L) & \text{if } CAC_{r,L} = A \\ ACS_{in} & \text{if } CAC_{r,L} = N \\ Join_x(ACS_{in}, Update_x(ACS_{in}, r, L)) & \text{if } CAC_{r,L} = U \vee CAC_{r,L} = U\text{-}N \end{cases}$$

where $Update_x^{ml}$ represents the multi-level version of the $Update_x$ functions presented earlier (§ 4.2). This definition can be inflected to $Update_{Must}^{ml}$, $Update_{Persistence}^{ml}$ or $Update_{May}^{ml}$ using the corresponding *Update* and *Join* functions of the Must, Persistence or May analysis respectively.

Termination of the analyses. The differences between the multi-level data cache analysis and the single-level data cache analysis are the $Update_x$ functions, $x \in \{Must, Persistence$ or $May\}$, for the different analyses. However these functions were proved to be monotonic in section 4.2. The proof in [7] thus holds for data caches as well. We need to demonstrate that the $Update_x^{ml}$ function is monotonic for the four possible values of CAC.

For an A access, $Update_x$ and $Update_x^{ml}$ behave identically. $Update_x$ being monotonic, $Update_x^{ml}$ is also monotonic. Considering a N access, $Update_x^{ml}$ is the identity function and so is monotonic. Finally, considering an U or U-N access, $Update_x^{ml}$ composes $Update_x$ and $Join_x$. These two functions are monotonic, so is their composition. Thus $Update_x^{ml}$ is monotonic which guarantees the termination of our analysis. □

4.4. WCET computation

CHMCs represent the worst-case behaviour of the cache given a memory reference. They are useful to compute the contribution of references to the WCET. This contribution can then be used in existing methods to compute the WCET. In our case, we focus on IPET based methods which estimates the WCET by solving an *Integer Linear Programming* (ILP) problem [10].

The timing of the memory reference r, with regards to the data caches, is divided in two parts. $first$ and $next$ respectively distinguish the first and successive iterations of loops. We define $COST_first(r)$ and $COST_next(r)$ as the respective contribution to the WCET of memory reference r for the first and successive iterations of the loop in which the reference is enclosed, if any[3]. If no loop encloses r, $COST_first(r)$ will be implicitly preferred, by the ILP solver, over $COST_next(r)$ as the cost of r. Indeed $COST_first(r)$ accounts for all first misses of memory reference r and $COST_first(r) \geq COST_next(r)$.

With $freq_r$ a variable computed in the context of IPET analysis and representing the execution frequency of r along the worst-case execution path of the task, the following constraints are defined: $freq_r = freq_{first,r} + freq_{next,r}$ and $freq_{first,r} \leq 1$. The WCET contribution of the reference r with regards to data caches is then defined as:

$$WCET_data_contribution(r) = COST_first(r) \times freq_{first,r} + COST_next(r) \times freq_{next,r}$$

As we focus on an architecture without timing anomalies, it is safe to consider that NC references behave like AM ones, which are the worst-case on our architecture. Therefore, U accesses to a cache level behave as A ones, from the timing analysis considering data caches point of view[4]. We define $always_contribute(r)$ and $never_contribute(r)$ as the sets of memory hierarchy levels which respectively always or never contribute to the execution latency of memory reference r, with $M = N + 1$ the main memory in the memory hierarchy:

$$never_contribute(r) = \{L \mid 1 \leq L \leq M \wedge CAC_{r,L} = N\}$$

$$always_contribute(r) = \{L \mid 1 \leq L \leq M \wedge (CAC_{r,L} = A \vee CAC_{r,L} = U)\}$$

One option to deal with a FM classification for reference r on cache level L would be to consider, for every execution of r, that cache level $L+1$ is accessed: $CAC_{r,L} =$ U-N $\Rightarrow L \in always_contribute(r)$. However, it might be overly pessimistic with regards to the semantic of the FM and the de facto inherited U-N classifications. We need additional notations to depict this behaviour and tighten r contribution to the WCET.

Given a memory reference r and a cache level L, let $memory_blocks_{r,L}$ be the set of r target memory blocks on cache level L, as computed by the address analysis. This information is computed

[3]Remember that r is contextual and attached to a unique instruction. A memory reference tied to the same instruction, but another context may be enclosed in different loops.

[4]From the ACS computing point of view, they keep different meanings.

using the cache blocks size of cache level L. We only need to know a bound on the size of this set:$|memory_blocks_{r,L}| = \lceil \frac{Addr_range_r}{cache_block_size_L} \rceil$ with $Addr_range_r$ the size of the address range computed by the address analysis for memory reference r and $cache_block_size_L$ the size of level L cache blocks.

Furthermore, we define max_freq_r as the maximum statically computable execution frequency of reference r. Such an information is computed as the product of the maximum number of iterations of all loops containing r: $max_freq_r = \prod_{lo \in Loops(r)} max_iter_{lo}$, where $Loops(r)$ are the loop containing the memory reference r and max_iter_{lo}, the maximum iteration attribute for loop lo. Note that the whole task is itself considered to be a loop enclosing all memory references and whose $max_iter = 1$, ensuring that the max_freq attribute of an otherwise not enclosed in a loop memory reference is 1.

Each element of $memory_block_{r,L}$ produces at most one miss in the cache level L, the first time it is accessed, according to the definition of the FM classification. Thus, reference r will not produce more than $min(max_freq_r, |memory_blocks_{r,L}|)$ misses on cache level L.

Note that $min(freq_r, |memory_blocks_{r,L}|)$, with $freq_r$ the execution frequency of r on the worst-case execution path of the task, would be a tighter bound. However, this bound is required to compute $freq_r$ and vice versa thus leading to a chicken-and-egg problem.

Once these elements have been defined, we can bound the number of occurrences of memory reference r on cache level L:

$$max_occurrence(r, L) = \begin{cases} 0 & if\ L \in never_contribute(r) \\ max_freq_r & if\ L \in always_contribute(r) \\ min(|memory_blocks_{r,L-1}|, max_occurrence(r, L-1)) & if\ CHMC_{r,L-1} = FM \\ max_occurrence(r, L-1) & otherwise \end{cases}$$

Intuitively, if cache level L is never (respectively always) accessed by memory reference r, there cannot be any (respectively more than max_freq_r) occurrences of r on cache level L. Similarly, there cannot be more occurrences of r on cache level L than on cache level $L-1$. Finally, if $CHMC_{r,L-1} = FM$, only the first accesses to memory blocks belonging to $memory_blocks_{r,L-1}$ will occur on cache level L.

The definition of $COST_next(r)$ is pretty straightforward, as we only have to count the access latency of cache levels L which are guaranteed to always contribute to the timing of r. Other accesses are considered in $COST_first(r)$:

$$COST_next(r) = \begin{cases} \sum_{L \in\ always_contribute(r)} ACCESS_latency_L & \text{if r is a load of data} \\ STORE_latency & \text{if r is a store of data} \\ 0 & \text{otherwise} \end{cases}$$

$COST_first(r)$ is a bit harder to define. In some cases, we have a bound on the number of occurrences of a memory reference r on cache L. As previously stated, the solution of not considering these bounds might be unnecessary pessimistic. Therefore, we chose to use the $COST_first(r)$ to hold these additional latencies. It could be understood as considering all the possible first misses in the first execution of r, in addition to all the always accessed cache levels latencies:

$$COST_first(r) = \begin{cases} \sum_{L \in\ always_contribute(r)} ACCESS_latency_L + \sum_{CAC_{r,L} = \text{U-N}} ACCESS_latency_L \times max_occurrence(r, L) & \text{if r loads data} \\ STORE_latency & \text{if r stores data} \\ 0 & \text{otherwise} \end{cases}$$

5. Experimental results

5.1. Experimental setup

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization and with the default linker memory layout. The WCETs of tasks are computed by the Heptane timing analyser [1], more precisely its Implicit Path Enumeration Technique (IPET). The timing of memory references with regards to data caches is evaluated using the method introduced in Section 4.4. The analysis is context sensitive (functions are analysed in each different calling context). To separate the effects of caches from those of the other parts of the processor micro-architecture, WCET estimation only takes into account the contribution of caches to the WCET. The effects of other architectural features are not considered. In particular, timing anomalies caused by interactions between caches and pipelines, as defined in [12] are disregarded. The cache classification *not-classified* is thus assumed to have the same worst-case behaviour as *always-miss* during the WCET computation in our experiments. The cache analysis starts with an empty cache state.

Name	Description	Code size (bytes)	Data size (bytes)	Bss size (bytes)	Stack size (bytes)
crc	Cyclic redundancy check computation	1432	272	770	72
fft	Fast Fourier Transform	3536	112	128	288
insertsort	Insertsort of an array of 11 integers	472	0	44	16
jfdctint	Fast Discrete Cosine Transform	3040	0	512	104
ludcmp	Simultaneous Linear Equations by LU Decomposition	2868	16	20800	920
matmult	Product of two 20x20 integer matrixes	1048	0	4804	96
minver	Inversion of floating point 3x3 matrix	4408	152	520	128
ns	Search in a multi-dimensional array	600	5000	0	48
qurt	Root computation of quadratic equations	1928	72	60	152
sqrt	Square root function implemented by Taylor series	544	24	0	88
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900	32	290	88

Table 2: Benchmark characteristics

Benchmarks. The experiments were conducted on a subset of the benchmarks maintained by Mälardalen WCET research group[5]. Table 2 summarizes the characteristics characteristics (size in bytes of sections *text*, *data*, *bss* (unitialized data) and *stack*).

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a 4-way L1 data cache of 1KB with a cache block size of 32B and a 8-way L2 data cache of 4KB with a cache block size of 32B. A perfect private instruction cache, with an access latency of one cycle, is assumed. All caches are implementing a LRU replacement policy. Latencies of 1 cycle (respectively 10 and 100 cycles) are assumed for the L1 cache (respectively the L2 cache and the main memory). Upon store instructions a latency of 150 cycles is assumed to update the whole memory hierarchy.

5.2. Considering the cache hierarchy

To evaluate the interest of considering the whole cache hierarchy, two configurations are compared in Table 3. In the first configuration, only the L1 cache is considered and all accesses to the L2 cache are defined as misses ($WCET_{L1}$, column 1). In the other configuration ($WCET_{L1\&L2}$, column 2), both the L1 and the L2 caches are analysed using our multi-level static cache analysis (§ 4.3). Table 3 presents these results for the two cases using the tasks WCET as computed by our tool and based on our multi-level static analysis. The presented WCET are expressed in cycles.

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

The third column of Table 3 represents the improvement, for each analysed task, attributable to the consideration of the second cache level of the hierarchy: $\frac{WCET_{L1} - WCET_{L1\&L2}}{WCET_{L1}}$.

Benchmark	WCET considering only a L1 cache (cycles)	WCET considering both L1 and L2 cache (cycles)	Improvement considering the L2 cache
crc	9373240	9119240	2.7 %
fft	14068000	6320470	55.07 %
insertsort	456911	456911	0 %
jfdctint	597239	464239	22.26 %
ludcmp	1778460	1778460	0 %
matmult	24446200	24446200	0 %
minver	328506	303626	7.57 %
ns	861575	861575	0 %
qurt	622711	520531	16.4 %
sqrt	94509	94509	0 %
statemate	1303760	1129380	13.37 %

Table 3: Evaluation of the static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)).

Considering the L2 cache using our multi-level static analysis, may lead up to 55.07% improvement for the *fft* task. In average, the results are still interesting 10.67% (6.23% whithout *fft*) with the exceptions of a subset of tasks not taking any benefit in the consideration of the L2 cache (*insertsort*, *ludcmp*, *matmult*, *ns* and *sqrt*).

The issues raised by these tasks are twofold. On the one hand, tasks like *sqrt* and *insertsort* use little data. This small working set fits in the L1 cache. Thus, the L2 cache is only accessed upon the very first misses. On the other hand are tasks like *ludcmp*, *matmult* and *ns* accessing large amount of data that fit neither in the L1 nor the L2 cache. *ludcmp* accesses a small portion of a large array of floats, which probably shows temporal locality but, as our address analysis returns the whole array range, this precision is lost. Concerning *matmult* and *ns*, they consist of loop nests running through big arrays. Again, there is temporal locality between the different iterations of the most nested loop. However, this locality is not captured as the FM classification applies to the outermost loop in such cases, and thus consider the whole array as being persistent or not.

Other tasks exhibit such small data set but, due to the analysis pessimism, some memory blocks might be considered as shifted outside the L1. However, they are detected as persistent in the L2 (*fft*, *jfdctint*, *minver*, *qurt*, *statemate* and to a lesser extent *crc*).

6. Conclusion

In this paper we presented a multi-level data cache analysis. This approach considers LRU set-associative non-inclusive caches implementing the *write-through*, *write-no-allocate* update policies. Results shows that considering the whole cache hierarchy is interesting with a computed WCET contribution of data caches being in average 10.67% smaller than the contribution considering only the first level of cache. Furthermore, the computation time is fairly reasonable, results for the eleven presented benchmarks being computed in less than a couple of minutes.

Keeping the same study context, future researches include the definition of less pessimistic constraints on the number of misses or improving the precision of the different analyses to handle tighter classifications (e.g. a classification per accessed block instead of per memory reference). Extending the analysis to other data cache configurations, whether speaking of replacement policies (e.g. pseudo-LRU) or update policies (e.g. write-back or write-allocate) might be the subject of other studies. Finally, the extension of the analysis context to handle multi-tasking systems or multi-core architectures is left as future works.

References

[1] COLIN, A., AND PUAUT, I. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)* (Delft, The Netherlands, June 2001), pp. 37–44.

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

[3] FERDINAND, C., HECKMANN, R., LANGENBACH, M., MARTIN, F., SCHMIDT, M., THEILING, H., THESING, S., AND WILHELM, R. Reliable and precise WCET determination for real-life processor. In *EM-SOFT '01: Proceedings of the First International Workshop on Embedded Software* (Tahoe City, CA, USA, Oct. 2001), vol. 2211, pp. 469–485.

[4] FERDINAND, C., AND WILHELM, R. On predicting data cache behavior for real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (1998), Springer-Verlag, pp. 16–30.

[5] FERDINAND, C., AND WILHELM, R. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst. 17*, 2-3 (1999), 131–181.

[6] HARDY, D., AND PUAUT, I. Predictable code and data paging for real time systems. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems* (2008), IEEE Computer Society, pp. 266–275.

[7] HARDY, D., AND PUAUT, I. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium* (2008), IEEE Computer Society, pp. 456–466.

[8] HECKMANN, R., LANGENBACH, M., THESING, S., AND WILHELM, R. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, n7* (2003).

[9] LI, X., ROYCHOUDHURY, A., AND MITRA, T. Modeling out-of-order processors for WCET analysis. *Real-Time Syst. 34*, 3 (2006), 195–227.

[10] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation* (1995), ACM, pp. 456–461.

[11] LUNDQVIST, T., AND STENSTRÖM, P. A method to improve the estimated worst-case performance of data caching. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications* (1999), IEEE Computer Society, p. 255.

[12] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium* (1999), IEEE Computer Society, p. 12.

[13] MUELLER, F. Static cache simulation and its applications. PhD thesis, 1994.

[14] MUELLER, F. Timing analysis for instruction caches. 217–247.

[15] PUAUT, I. WCET-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)* (Dresden, Germany, July 2006).

[16] RAMAPRASAD, H., AND MUELLER, F. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium* (2005), IEEE Computer Society, pp. 148–157.

[17] SEN, R., AND SRIKANT, Y. N. WCET estimation for executables in the presence of data caches. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software* (2007), ACM, pp. 203–212.

[18] THEILING, H., FERDINAND, C., AND WILHELM, R. Fast and precise WCET prediction by separated cache and path analyses. 157–179.

[19] VERA, X., LISPER, B., AND XUE, J. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium* (Cancun, Mexico, 2003).

[20] VERA, X., LISPER, B., AND XUE, J. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys. 7*, 1 (2007), 1–38.

[21] VERA, X., AND XUE, J. Let's study whole-program cache behaviour analytically. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (2002), IEEE Computer Society, p. 175.

[22] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), ACM, pp. 30–44.

WCET-AWARE SOFTWARE BASED CACHE PARTITIONING FOR MULTI-TASK REAL-TIME SYSTEMS [1]

Sascha Plazar[2], Paul Lokuciejewski[2], Peter Marwedel[2]

Abstract

Caches are a source of unpredictability since it is very difficult to predict if a memory access results in a cache hit or miss. In systems running multiple tasks steered by a preempting scheduler, it is even impossible to determine the cache behavior since interrupt-driven schedulers lead to unknown points of time for context switches. Partitioned caches are already used in multi-task environments to increase the cache hit ratio by avoiding mutual eviction of tasks from the cache.

For real-time systems, the upper bound of the execution time is one of the most important metrics, called the Worst-Case Execution Time (WCET). In this paper, we use partitioning of instruction caches as a technique to achieve tighter WCET estimations since tasks can not be evicted from their partition by other tasks. We propose a novel WCET-aware cache partitioning algorithm, which determines the optimal partition size for each task with focus on decreasing the system's WCET for a given set of possible partition sizes. Employing this algorithm, we are able to decrease the WCET depending on the number of tasks in a set by up to 34%. On average, reductions between 12% and 19% can be achieved.

1 Introduction

Embedded systems often operate as hard real-time systems which have to meet hard timing constraints. For these systems, it is mandatory to know the upper bound of the execution time for each task and possible input data. This bound is called *Worst-Case Execution Time*.

Caches have become popular to bridge the gap between high processor and low memory performance. The latency for an access to a certain memory address highly depends on the content of the cache. If an instruction to be fetched already resides in the cache, then a so called *cache hit* occurs and the fetch can be usually performed within one cycle. Otherwise, it results in a *cache miss*. The desired address has to be fetched from the slow main memory (e.g. Flash) leading to penalty cycles depending on the processor and memory architecture.

It is hard to determine statically if an arbitrary memory access results in a cache hit or a cache miss. However, caches are used in real-time systems because they can drastically speed up the execution of programs. Hence, a lot of effort has been successfully put into research to make sound prediction about the worst-case cache performace during a program's execution. AbsInt's *aiT* [2] is a tool that performs static analyses on binary programs to predict their cache behavior and WCET.

[1] The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

[2] Computer Science 12 — TU Dortmund University — D-44221 Dortmund, Germany — FirstName.LastName@tu-dortmund.de

In environments with preemptive schedulers running more than one task, it is impossible to make any assumption about the memory access patterns. This is mainly caused by interrupt-driven scheduling algorithms causing context switches at unknown points of time. Thus, the program's state in not known at which a context switch occurs. It is also unknown at which address the execution of a program continues, hence it is unknown which line of the cache is evicted next. An unknown cache behavior forces to assume a cache miss for every memory access implying a highly overestimated systems overall WCET. As a consequence, the underlying system has to be oversized to meet real-time constraints resulting in higher costs for hardware.

We adapt an existing technique called software based cache partitioning [16] to make the instruction cache (*I-cache*) behavior more predictable. This can be guaranteed since every task has its own cache partition from which it can not be evicted by another task. Our novel WCET-aware cache partitioning aims at selecting the optimal partition size for each task of a set to achieve the optimal WCET minimization. The main contributions of this paper are as follows:

1. Compared to existing approaches which focus on minimization of average-case execution times, our WCET-aware cache partitioning explicitly evaluates WCET data as metric for optimization.

2. In contrast to previous works which presented theories to partition a cache in software, our approach comprises a fully functional implementation of a cache partitioning method.

3. We show that our ILP-based WCET-aware cache partitioning is effective in minimizing a system's WCET and outperforms existing algorithms.

The paper is organized as follows: In the next section, we present related work. Existing techniques to partition a cache as well as our new algorithm are explained in Section 3. Section 4 introduces the compiler WCC used to integrate our novel cache partitioning algorithm. An evaluation of the performance which is achieved by our WCET-aware cache partitioning, is presented in Section 5. Finally, we conclude our work and give a brief overview of future work.

2 Related Work

The papers [16, 5, 15] present different techniques to exploit cache partitioning realized either in hardware or in software. In contrast to our work, these implementations either do not take the impact on the WCET into account or do not employ the WCET as the key metric for optimization which leads to suboptimal or even degraded results. In [16], the author presents ideas for compiler support for software based cache partitioning which serves as basis for the partitioning techniques presented in this paper. Compared to the work in this paper, a functional implementation or impacts on the WCET are not shown. In [5], a hardware extension for caches is proposed to realize a dynamic partitioning through a fine grained control of the replacement policy via software. Access to the cache can be restricted to a subset of the target cache set which is called columnization. For homogeneous on-chip multi-processor systems sharing a unified set-associative cache, [15] presents partitioning schemes based on associativity and sets.

A combination of locking and partitioning of shared caches on multi-core architectures is researched in [18] to guarantee a predictable system behavior. Even tough the authors evaluate the impact of their caching schemes on the worst-case application performance, their algorithms are not WCET-aware. Kim et al. [11] developed an energy efficient partitioned cache architecture to reduce the energy per access. A partitioned L1-cache is used to access only one sub-cache for every instruction fetch leading to dynamic energy reduction since other sub-caches are not accessed.

The authors of [4] show the implications of code expanding optimizations on instruction cache design. Different types of optimizations and their influence on different cache sizes are evaluated. [12] gives an overview of cache optimization techniques and cache-aware numerical algorithms. It focuses on the bottleneck memory interface which often limits the performance of numerical algorithms.

Puaut et al. counteract the problem of unpredictability with locked instruction caches in multi-task real-time systems. They propose two low complexity algorithms for cache content selection in [17]. A drawback of statically locking the cache content is that the dynamic behavior of the cache gets lost. Code is no more automatically loaded into the cache, thus code which is not locked into the cache can not profit from it anymore.

Vera et al. [22] combine cache partitioning, dynamic cache locking and static cache analysis of data caches to achieve predictability in preemptive systems. This eliminates overestimation and allows to approximate the worst-case memory performance.

Lokuciejewski et. al rearrange the orders of procedures in main memory to exploit locality in the control flow leading to a higher cache performance [13]. Worst-case calling frequencies serve as metrics for WCET minimization but multi-task sets are not supported.

3 WCET-aware Cache Partitioning

Caches have become popular to bridge the growing gap between processor and memory performance since they are transparent from the programmer's point of view. Unfortunately, caches are a source of unpredictability because it is very difficult to determine if a memory access results in a cache hit or a cache miss. Static analysis is a technique to predict the behavior of the cache [19] and make sound prediction about the WCET of a program which allows the profitable usage of caches in real-time systems running a single task.

In general, real-time systems consist of more than one task which makes it often impossible to determine the worst-case cache behavior. Due to interrupt driven schedulers, points of time for context switches can not be statically determined. Thus, it is not predictable which memory address is fetched from the next task being executed and one can not make proven assumptions which cache line is replaced by such an unknown memory access. Due to this fact, every memory access has to be treated as a cache miss leading to a highly overestimated WCET caused by an underestimated cache performance.

In a normally operating cache, each task can be mapped into any cache line depending on its memory usage. To overcome this situation, partitioned caches are recommended in literature [16, 5, 15]. Tasks in a system with partitioned caches can only evict cache lines residing in the partition they are assigned to. Reducing the prediction problem of replaced cache lines to one task with its own cache partition, allows the application of well known single task approaches for WCET- and cache performance estimation. The overall execution time of a task set is then composed of the execution time of the single tasks with a certain partition size and the overhead required for scheduling including additional time for context switches.

Infineon's TriCore architecture does not support partitioned caches in hardware so that partitioning has to be done in software. The following section describes the basics of software based cache partitioning schemes applied in our WCET-aware cache partitioning algorithms. In Section 3.2, a heuristic approach is applied to determine the partitioning based on the tasks' sizes. Section 3.3 presents our novel algorithm for selecting an optimal partition size w.r.t. the overall WCET of a system.

3.1 Software based Cache Partitioning

The author in [16] presents a theory to integrate software based cache partitioning into a compiler toolchain without an existing implementation. Code should be scattered over the address space so that tasks are mapped to certain cache lines. Therefore, all tasks have to be linked together in one monolithic binary and a possible free space between several parts has to be filled with *NOPs*. Partitioning for data caches involves code transformation of data references.

The theory to exactly position code in the address space to map it into certain cache lines is picked up here, but a completely different technique is applied to achieve such a distribution. We restrict ourselves to partitioning of I-caches, thus only software based partitioning of code using the new technique is discussed. However, a partitioning of data caches w.r.t. WCET decrease is straightforward using a modified version of our algorithm.

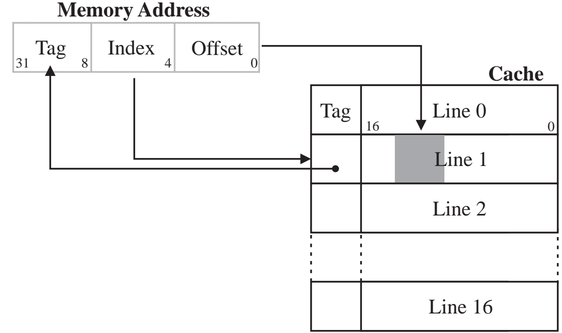


Figure 1: Addressing of cache content

For the sake of simplicity, in the following we assume a direct-mapped cache. Way associative caches can be partitioned as well: The desired partition size has to be divided by the degree d of associativity since any particular address in main memory can be mapped in one of d locations in the cache. In this case, predictable replacement policies (e.g. *last recently used* for TriCore) are allowed to enable static WCET-analysis. However the replacement policy has no influence on the partitioning procedure.

Assuming a very small cache with $S = 256$ bytes capacity divided into $l = 16$ cache lines, results in a cache line size of $s = 16$ bytes. When an access to a cached memory address is performed, the address is split into a tag, an index, and an offset part. Our example in Figure 1 shows the 4 offset bits addressing the content inside a cache line, whereas 4 index bits select a cache line. The remaining address bits form the tag which is stored in conjunction with the cache line. The tag bits have to be compared for every cache access since arbitrary memory addresses with the same index bits can be loaded into the same line.

To partition a cache, it has to be ensured that a task assigned to a certain partition only allocates memory addresses with index bits belonging to this partition. For an instruction cache divided into two partitions of 128 bytes, one partition ranges from cache line 0 to line 7 and the second one from line 8 up to 15. If a task T_1 is assigned to the first partition, each occupied memory address must have index bits ranging from `000b` up to `111b` accessing the cache lines 0 to 7 and arbitrary offset bits. Together, index and offset bits correspond to memory addresses modulo cache size ranging from `0x00` to `0x7f`. A task T_2 assigned to the second partition has to be restricted to cover only memory addresses modulo cache size from `0x80` up to `0xff`.

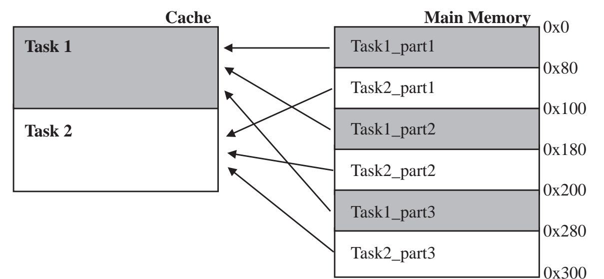


Figure 2: Mapping of tasks to cache lines

Tasks exceeding the size of the partition they are mapped to, have to be split and scattered over the address space. Figure 2 illustrates the partitioning for tasks T_1 and T_2 into such 128 bytes portions and the distribution of these portions over the main memory. Task T_1 is allocated to portions which are mapped to the first half of the cache since all occupied memory addresses modulo cache size

range from 0-127. The same has to meet for task T_2 occupying memory addresses modulo cache size ranging from 128-255.

Obviously, partitioning does not depend on the cache line size since a contiguous part of the memory is always mapped into the same amount of cache memory. Only the atomic size for composing partitions is equal to the cache line size, thus the partition size must be a multiple thereof.

WCC's workflow employs the linker to achieve such a distribution of code over the address space. Individual linker scripts are used (compare Listing 1 for the aforementioned example) with relocation information for every task and its portions it is divided into. For linker basics refer to [3].

The output section `.text`, to be created in the output binary (line 1), is aligned to a memory address which is a multiple of the cache size to ensure that the mapping starts at cache line 0. Line 3 allocates the assembly input section `.task1_part1` at the beginning of the `.text` output section, thus the beginning of the cache. The content of this section must not exceed 128 bytes since line 4 sets the relocation counter to the address 128 bytes beyond the start address, which is mapped into the first line of the second cache half. Line 5 accomplishes the relocation of section `.task2_part1` to the new address. The other sections are mapped in the same manner.

```
.text: {                              1
  _text_begin = .;                    2
  *(.task_part1)                      3
  . = _text_begin + 0x80;             4
  *(.task2_part1)                     5
  . = _text_begin + 0x100;            6
  *(.task1_part2)                     7
  . = _text_begin + 0x180;            8
  *(.task2_part2)                     9
  . = _text_begin + 0x280;            10
  *(.task2_part3)                     11
} > PFLASH-C                          12
```

Listing 1: Linker script example

On the assembly level, each code portion which should be mapped to a partition, has to be attached to its own linker section to cause a relocation by the linker; e.g. `.task_part1` for the first 128 bytes memory partition of task T_1. To restore the original control flow, every memory partition has to be terminated by an additional unconditional branch to the next memory partition of the task unless the last instruction of this block already performs an unconditional transfer of control.

For further jump corrections required by growing displacements of jump targets and jump sources refer to [16].

3.2 Size-driven Partition Size Selection

The author in [16] propose to select a task's partition size depending on its size relative to the size of the complete task set. For our example, a task set with $m = 4$ tasks $T_1 - T_4$ having a size of $s(T_1) = 128$ bytes, $s(T_2) = 256$ bytes, $s(T_3) = 512$ bytes and $s(T_4) = 128$ bytes should be assumed. Hence, the complete task set has an overall code size of 1 kB, whereas we use the assumed cache from the previous section with a capacity of $S = 256$ bytes.

According to its size, task T_i's partition size computes as follows:

$$p(T_i) = \frac{s(T_i)}{\sum_{j=1}^{n} s(T_j)} * S_{cache} \tag{1}$$

e.g. T_1 is assigned to a partition with $128\ bytes/1024\ bytes = 1/8$ of the cache size. Accordingly the assigned partition sizes are: $p(T_1) = 32$ bytes, $p(T_2) = 64$ bytes, $p(T_3) = 128$ bytes and $p(T_4) = 32$ bytes.

3.3 WCET-driven Partition Size Selection

The size of a cache may have a drastic influence on the performance of a task or an embedded system. Caches with sufficient size can decrease the runtime of a program whereas undersized caches can lead

to a degraded performance due to a high cache miss ratio. Hence, it is essential to choose the optimal partition size for every task in order to achieve the highest possible decrease of the system's overall WCET.

Current approaches select the partition size depending on the code size or a tasks priority [16, 18]. They aim at improving a system's predictability and examine the impact of partitioning on the WCET but do not explicit aim at minimizing its WCET.

In this section, we present our novel approach to find the optimal partition sizes for a set of tasks w.r.t. the lowest overall WCET of a system. We use integer linear programming (*ILP*) to select the partition size for each task from a given set of possible partition sizes.

We assume that there is a set of m tasks which are scheduled periodically. There is a schedule interval within each task $T_i \in T$ is executed exactly c_i times, which is repeated continuously. The length of this interval is the least common multiple of the m tasks' periods. Furthermore, we assume a set P of given partition sizes with $|P| = n$ partitions, e.g. $P = \{0, 128, 256, 512, 1024\}$ measured in bytes. Let x_{ij} be a binary decision variable determining if task T_i is assigned to a partition with size $p_j \in P$:

$$x_{ij} = \begin{cases} 1, \text{ if } T_i \text{ assigned to } p_j \\ 0, \text{ else} \end{cases}$$

To ensure that a task is assigned to exactly one partition, the following m constraints have to be met:

$$\forall i = 1..m : \sum_{j=1}^{n} x_{ij} = 1 \tag{2}$$

$WCET_{ij}$ is T_i's WCET for a single execution if assigned to partition p_j, then the WCET for a single task T_i is calculated as follows:

$$WCET(T_i) = \sum_{j=1}^{n} x_{ij} * WCET_{ij}$$

Since we focus on WCET minimization, we define the cost function to be minimized for the whole task set:

$$WCET = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} * c_i * WCET_{ij} \tag{3}$$

To keep track of the limited cache size S we introduce another constraint:

$$\sum_{j=1}^{m} \sum_{i=1}^{n} x_{ij} * p_j \leq S \tag{4}$$

Using equations 2 to 4, we are able to set up the cost function and $m + 1$ constraints as input for

Input: Set of tasks T, set of partition sizes P, execution counts C, cache size S
Output: Set of partitioned tasks T

```
1  begin
2    for t_i ∈ T do
3      for p_j ∈ P do
4        partitionTask( t_i, p_j);
5        WCET_ij = determineWCET( t_i );
6        WCET = WCET ∪ WCET_ij;
7      end
8    end
9    ilp = setupEquat( T, P, WCET, C, S);
10   X = solveILP( ilp);
11   forall x_ij ∈ X : x_ij = 1 do
12     partitionTask( t_i, p_j);
13   end
14   return T;
15 end
```

Algorithm 1: **Pseudo code of cache partitioning algorithm**

an ILP solver like *lp_solve* [1] or *CPLEX* [10]. After solving the set of linear equations, the minimized WCET and all variables $x_{ij} = 1$, representing the optimal partition sizes for all tasks, are known.

The number of necessary WCET analyses depends on the number of tasks and the number of possible partition sizes which have to be taken into account: $\#Analyses_{WCET} = |T| * |P| = m * n$

To determine the WCETs, to set up all equations and to apply partitioning, Algorithm 1 is employed. A given task set, the instruction cache size and a set of possible partition sizes for the tasks are required as input data. The algorithm iterates over all tasks (line 2) and temporary partitions each task (line 3 to 4) for all given partition sizes. Subsequently, the WCET for the partitioned task is determined invoking AbsInt's static analyzer aiT (line 5). Exploiting the information about tasks, partition sizes, cache size and gathered WCETs, an ILP model is generated regarding equations 3 to 4 (line 8) and solved in line 9.

Afterwards, the set X includes exactly one decision variable x_{ij} per task T_i with the value 1 whereas p_j is T_i's optimal partition size w.r.t. minimization of the system's WCET. Finally, in lines 11 to 12 a software-based partitioning of each task with its optimal partition size, as described in Section 3.1, is performed.

4 Workflow

Software based cache partitioning needs the support of an underlying compiler to collect WCET-data, perform the required code modifications and scatter the code over the address space. We employ our WCET-aware C compiler framework, called *WCC* [8], intended to develop various high- and low-level optimizations. WCC is a compiler for the Infineon TriCore TC1796 processor coupling AbsInt's static WCET analyzer *aiT* [2] which provides WCET data that is imported into the compiler backend and made accessible for optimizations.

Figure 3 depicts WCC's internal structure reading the tasks of a set in the form of ANSI-C source files with user annotations for loop bounds and recursion depths, called *flow facts*. These source files are parsed and transformed into our high-level intermediate representation (*IR*) *ICD-C* [6]. Each task in a set is represented by its own IR.

In the next step, the *LLIR Code Selector* translates the high-level IRs into low-level IRs called *ICD-LLIR* [7]. On these TriCore TC1796 specific assembly level IRs, the software based cache partitioning can be performed. To enable such a WCET-aware optimization, AbsInt's *aiT* is employed to perform static WCET analyses on the low-level IRs. Therefore, mandatory information about loop bounds and recursion depth is supplied by flow fact annotations.

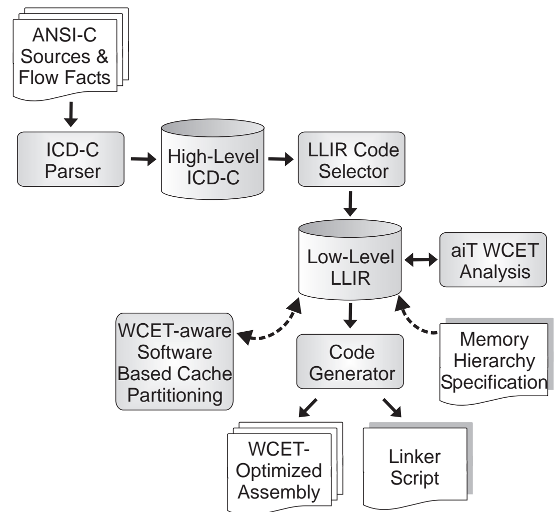


Figure 3: Workflow of the WCET-aware C compiler WCC

Optimizations exploiting memory hierarchies such as our novel software based cache partitioning require detailed information about available memories, their sizes and access times. For this purpose, WCC integrates a detailed memory hierarchy specification available on *ICD-LLIR* level.

Finally, WCC emits WCET-optimized assembly files and generates suitable binaries using a linker script reflecting the utilized internal memory layout.

5 Evaluation

This section compares the capability of our WCET-driven cache partitioning to existing partition size selection heuristic based on tasks sizes. We use different task sets from media and real-time benchmark suites to evaluate our optimization on computing algorithms typically found in the embedded systems domain. Namely, tasks from the suites *DSPstone* [21], *MRTC* [14] and *UTDSP* [20] are evaluated. WCC supports the Infineon TriCore architecture whose implementation in form of the TC1796 processor is employed for the evaluation. The processor integrates a 16 kB 2-way set associative I-cache with 32 bytes cache line size.

Overall, the used benchmark suites include 101 benchmarks so that we have to limit to a subset of tasks for cache partitioning. For lack of specialized benchmarks suites, sets of tasks stemming from the mentioned benchmark suite, as proposed in [9], are generated and compiled with the optimization level `-O3`. Using these sets, we benchmark the capability of decreasing the WCET achieved by standard partitioning algorithms compared to our WCET-aware approach.

Different numbers of tasks (5, 10, 15) in a set are checked to determine their effect on the WCET. To gather presentable results, we compute the average of 100 sets of randomly selected tasks for each considered cache sizes and the differing task set sizes. Seven cache sizes with the power of two are taken into account, ranging from 256 bytes up to 16 kB. Thus, the overall number of ILPs for every benchmark suite, which has to be generated and solved, is:

$$|ILPs| = 3 * 100 * 7 = 2100$$

Due to the fact that we do not take scheduling into account for benchmarking, the tasks execution frequencies c_i (cf. equation (3.3)) are set to one, thus, the system's WCET is composed of the task's WCETs for a single execution.

Figure 4 shows the relative WCETs for the benchmark suite *DSPstone Floating Point*, achieved by our novel optimization presented in Section 3.3 as percentage of the WCET achieved by the standard heuristic presented in Section 3.2. The nominal sizes of the task sets range on average from 1.5 kB for 5 tasks up to 5 kB for 15 tasks. Substantial WCET reductions can only be obtained for smaller caches of up to 1 kB since almost all tasks fit into the cache from 4 kB on. There, WCET reductions between 4% and 33% can be observed. In general, larger task sets result in higher optimization potential for all cache sizes.

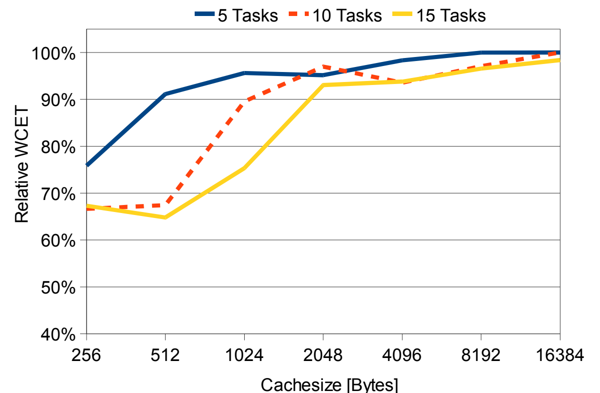


Figure 4: Optimized WCET for DSPstone Floating Point relative to standard approach

Figure 5 depicts the average WCET for the *MRTC* benchmark suite. The average code size of the generated task sets is comparatively large with 6 kB for 5 tasks, 12 kB for 10 tasks and 19 kB for 15 tasks. Hence, there is more potential to find a better distribution of partition sizes. This can be seen in a nearly

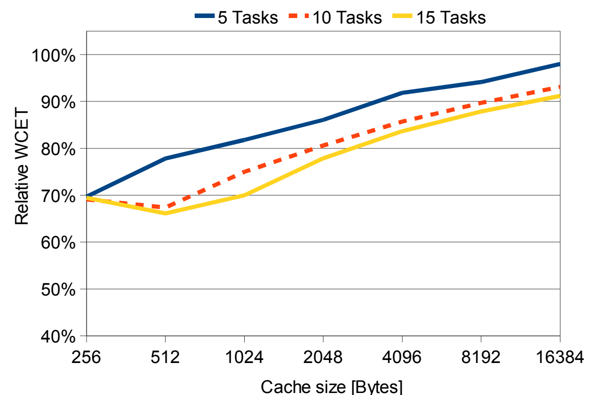


Figure 5: Optimized WCET for MRTC relative to standard approach

linear correlation of the optimization potential and the quotient of task set size and cache size. For 5 tasks in a set, WCET reductions up to 30% can be gained. 10 tasks per set have a higher optimization potential, so that 7% to 31% decrease of WCET can be observed. Optimizing the sets of 15 tasks, 9% up to 31% lower WCETs can be achieved.

A similar situation can be observed in Figure 6 for the *UTDSP* benchmark suite. The average code sizes for the task sets range from 9 kB to 27 kB. This leads to an optimization potential of 4% for a 5 task set completely fitting into the cache and 17% up to 36% for a 15 task set especially for small cache sizes. For this benchmark suite, the same behavior can be observed: for smaller cache sizes and larger code sizes our algorithm achieves better results compared to the standard approach.

Using caches larger than 16 kB, our algorithm is not able to achieve better or only marginal better results than if the standard method from section 3.2 is applied. This comes from the fact that mostly there is no optimization potential if all tasks fit into the cache. For realistic applications, the cache would be much smaller than the amount of code. There is also no case where the standard algorithm performs better than our approach since we use ILP models to always obtain the optimal partition size distribution.

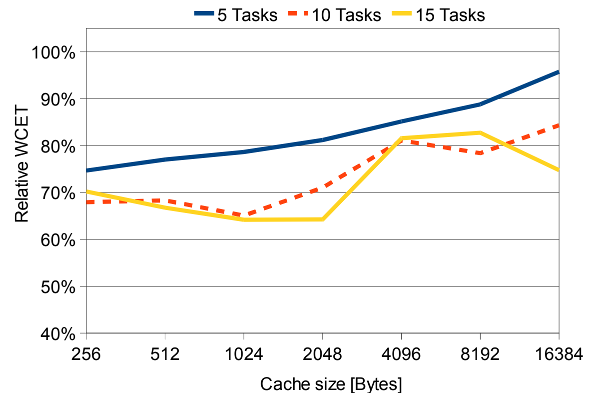


Figure 6: Optimized WCET for UTDSP relative to standard approach

Compilation Time

To consider compilation and optimization time on the host system, we utilize an Intel Xeon X3220 (2.40 GHz). A complete toolchain iteration is decomposed into the three phases compilation, WCET analysis, and optimization. The stage WCET analysis comprises all aiT invocations necessary to compute the tasks' WCETs for possible partition sizes.

The time required for a combined compilation and optimization phase ranges from less than one second (*fir* from MRTC) to 30 seconds for *adpcm* from UTDSP. Compared to this, the duration for performing static WCET analyses used for construction of an ILP is significantly higher with up to 10 hours .

6 Conclusions and Future Work

In this paper, we show how to exploit software based cache partitioning to improve the predictability of worst-case cache behavior in focus of multi-task real-time systems. Employing partitioned caches, every task has its own cache area from which it can not be evicted by other tasks. We introduce a novel algorithm for WCET-aware software based cache partitioning in multi-task systems to achieve predictability of cache behavior. The linker is exploited to achieve a restriction of tasks to be mapped into certain cache lines. An ILP model, based on the tasks' WCETs for different partition sizes, is set up and solved to select the optimal partition size for each task w.r.t. minimizing the systems WCET.

The new technique was compared to simple partition size selection algorithms in order to demonstrate its potential. The results show that our algorithm always finds better combinations of tasks' partition sizes than the size-based approach. Inspecting small task sets, we are able to decrease the WCET up to 30% compared to the standard approach. Better results can be achieved for larger task sets with up to 33% WCET reduction.

On average, we were able to outperform the size-based algorithm by 12% for 5 tasks in a set, 16% for task sets with 10 tasks, and 19% considering tasks sets with 15 tasks.

In general, the larger the task sets (and by association the code sizes) are, the better the results. This means: the algorithm performs best for realistic examples and less well for small (more academic) examples.

In the future, we intend to extend our algorithm to support partitioning of data caches. This enables predictable assumptions for the worst-case behavior of data caches accessed by multiple tasks in embedded systems with preempting schedulers. Another goal is the tightly coupling of offline scheduling algorithm analyses to automatically prefer those tasks during optimization which miss their deadlines.

References

[1] lp_solve reference guide. `http://lpsolve.sourceforge.net/5.5/`. v. 5.5.0.14.

[2] ABSINT ANGEWANDTE INFORMATIK GMBH. Worst-Case Execution Time Analyzer aiT for TriCore. `http://www.absint.com/ait`.

[3] CHAMBERLAIN, S., AND TAYLOR, I. L. *Using ld*, 2000. Version 2.11.90, `http://www.skyfree.org/linux/references/ld.pdf`.

[4] CHEN, W. Y., CHANG, P. P., CONTE, T. M., AND HWU, W. W. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Trans. Comput. 42*, 9 (1993).

[5] CHIOU, D., RUDOLPH, L., DEVADAS, S., AND ANG, B. S. Dynamic Cache Partitioning via Columnization. In *Proceedings of DAC* (2000).

[6] ECKART, J., AND PYKA, R. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c`, 2009. Informatik Centrum Dortmund, Embedded Systems Profit Center.

[7] ECKART, J., AND PYKA, R. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir`, 2009. Informatik Centrum Dortmund, Embedded Systems Profit Center.

[8] FALK, H., LOKUCIEJEWSKI, P., AND THEILING, H. Design of a wcet-aware c compiler. In *Proceedings of WCET* (`http://ls12-www.cs.tu-dortmund.de//research/activities/wcc`, 2006).

[9] HARDY, D., AND PUAUT, I. WCET Analysis of Multi-Level Non-Inclusive Set-Associative Instruction Caches. In *Proceedings of RTSS* (2008).

[10] ILOG. CPLEX. `http://www.ilog.com/products/cplex`.

[11] KIM, C., CHUNG, S., AND JHON, C. An Energy-Efficient Partitioned Instruction Cache Architecture for Embedded Processors. *IEICE - Trans. Inf. Syst.*, 4 (2006).

[12] KOWARSCHIK, M., AND WEI, C. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies* (2003), Springer.

[13] LOKUCIEJEWSKI, P., FALK, H., AND MARWEDEL, P. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of ECRTS* (Prague/Czech R., 2008).

[14] MÄLARDALEN WCET RESEARCH GROUP. Mälardalen WCET benchmark suite. `http://www.mrtc.mdh.se/projects/wcet`, 2008.

[15] MOLNOS, A., HEIJLIGERS, M., COTOFANA, S. D., AND EIJNDHOVEN, J. Cache Partitioning Options for Compositional Multimedia Applications. In *Proceedings of ProRISC* (2004).

[16] MUELLER, F. Compiler Support for Software-Based Cache Partitioning. *SIGPLAN Not. 30*, 11 (1995).

[17] PUAUT, I., AND DECOTIGNY, D. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of RTSS* (Washington, DC, USA, 2002), IEEE Computer Society.

[18] SUHENDRA, V., AND MITRA, T. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proceedings of DAC* (New York, USA, 2008).

[19] THEILING, H., FERDINAND, C., AND WILHELM, R. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Syst. 18*, 2-3 (2000).

[20] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, 2008.

[21] V. ZIVOJNOVIC, J. MARTINEZ, C. S., AND MEYR, H. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of ICSPAT* (Dallas, TX, USA, 1994).

[22] VERA, X., LISPER, B., AND XUE, J. Data Caches in Multitasking Hard Real-Time Systems. In *Proceedings of RTSS* (Cancun, Mexico, 2003).

WORST-CASE TIMING ESTIMATION AND ARCHITECTURE EXPLORATION IN EARLY DESIGN PHASES

Stefana Nenova and Daniel Kästner [1]

Abstract

Selecting the right computing hardware and configuration at the beginning of an industrial project is an important and highly risky task, which is usually done without much tool support, based on experience gathered from previous projects. We present TimingExplorer - a tool to assist in the exploration of alternative system configurations in early design phases. It is based on AbsInt's aiT WCET Analyzer and provides a parameterizable core that represents a typical architecture of interest. TimingExplorer requires (representative) source code and enables its user to take an informed decision which processor configurations are best suited for his/her needs. A suite of TimingExplorers will facilitate the process of determining what processors to use and it will reduce the risk of timing problems becoming obvious only late in the development cycle and leading to a redesign of large parts of the system.

1. Introduction

Choosing a suitable processor configuration (core, memory, peripherals, etc.) for an automotive project at the beginning of the development is a challenge. In the high-volume market, choosing a too powerful configuration can lead to a serious waste of money. Choosing a configuration not powerful enough leads to severe changes late in the development cycle and might delay the delivery.

Currently to a great extent this risky decision is taken based on gut feeling and previous experience. The existing timing analysis techniques require executable code as well as a detailed model of the target processor for static analysis or actual hardware for measurements. This means that they can be applied only relatively late in the design, when code and hardware (models) are already far developed. Yet timing problems becoming apparent only in late design stages may require a costly re-iteration through earlier stages. Thus, we are striving for the possibility to perform timing analysis already in early design stages.

Existing techniques that are applicable at this stage include performance estimation and simulation (e.g., virtual system prototypes, instruction set simulators, etc.). It should be noted that these approaches use manually created test cases and therefore only offer partial coverage. As a result, corner cases that might lead to timing problems can easily be missed. Since we follow a completely different approach, we will not further review existing work in these areas.

Our goal is to provide a family of tools to assist in the exploration of alternative system configurations before committing to a specific solution. They should perform an automatic analysis, be efficient and provide 100% coverage on the analyzed software so that critical corner cases are automatically

[1] AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2280

considered. Using these tools, developers will be able to answer questions like "what would happen if I take a core like the ABCxxxx and add a small cache and a scratchpad memory in which I allocate the C-stack or a larger cache" or "what would be the overall effect of having an additional wait cycle if I choose a less expensive flash module", ... Such tools can be of enormous help when dimensioning hardware and will reduce the risk of a project delay due to timing issues.

The paper is structured as follows. We describe how TimingExplorer is to be used in Section 2. In Section 3 we present AbsInt's aiT WCET Analyzer and compare it to TimingExplorer in Section 4. Finally we discuss a potential integration of TimingExplorer in a system-level architecture exploration analysis in Section 5.

2. TimingExplorer Usage

In the early stage of an automotive project it has to be decided what processor configuration is best suited for the application. The ideal architecture is as cheap as possible, yet powerful enough so that all tasks can meet their deadlines. There are usually several processor cores that come into question, as well as a number of possible configurations for memory and peripherals. How do we choose the one with the right performance?

Our approach requires that (representative) source code of (representative) parts of the application is available. This code can come from previous releases of a product or can be generated from a model within a rapid prototyping development environment.

The way to proceed is as follows. The available source code is compiled and linked for each of the cores in question. In the case one uses a model-based design tool with an automatic code generator, this step corresponds to launching the suitable code generator on the model. Each resulting executable is then analyzed with the TimingExplorer for the corresponding core. The user has the possibility to specify memory layout and cache properties. The result of the analysis is an estimation of the WCET for each of the analyzed tasks given the processor configuration. To see how the tasks will behave on the same core, but with different memory layout, memory or cache properties, one simply has to rerun the analysis with the appropriate description of cache and memory. Finally, the estimated execution times can be compared and the most appropriate configuration can be chosen.

Let us consider three tasks that come from a previous release of the system. Having an idea of the behavior of the old version of the system and the extensions we want to add to it in the new version, we wonder whether a MPC565 core will be sufficient for the project or it is better to use the more powerful MPC5554 core that offers more memory and cache and a larger instruction set. This situation is depicted in Figure 1. The tools we need are *TimingExplorer for MPC5xx* (*TE*$_{5xx}$) and *TimingExplorer for MPC55xx* (*TE*$_{55xx}$) for assessing the timing behavior of the tasks on the MPC565 and MPC5554 core respectively. First we compile and link the sources for MPC5xx and MPC55xx, which results in the executables E_{5xx} and E_{55xx}. Then we write the configuration files C_{5xx} and C_{55xx} to describe the memory layout, the memory timing properties and the cache parameters when applicable for the configurations in question. Next we analyze each of the tasks T_1, T_2 and T_3 for each of the two configurations. To get an estimate for the WCET of the tasks on the MPC5554 core, we run *TE*$_{55xx}$ with inputs E_{55xx} and C_{55xx} for each of the tasks. We proceed similarly to obtain an estimate of the timing for the tasks on the MPC565 core. By comparing the resulting times (X_1, X_2, X_3) and (Y_1, Y_2, Y_3), we see that the speed and resources provided by the MPC565 core will not be sufficient for our needs, so we settle for the MPC5554 core.

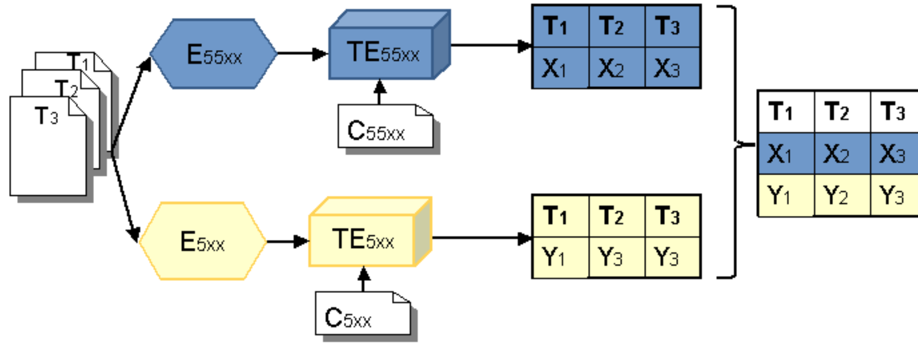


Figure 1. Usage of TimingExplorer to choose a suitable core

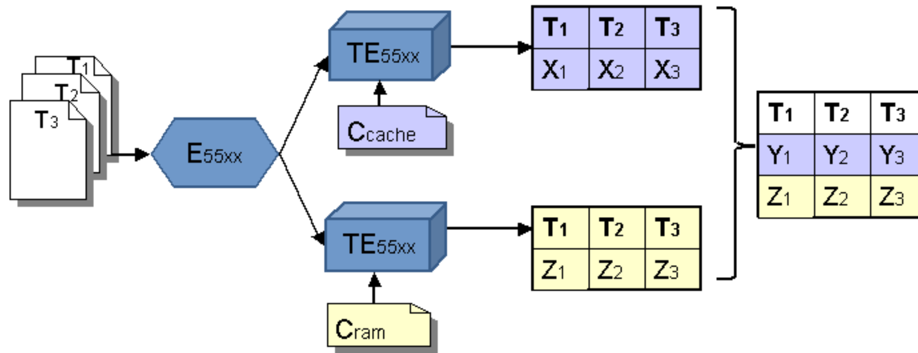


Figure 2. Usage of TimingExplorer to choose a suitable cache and memory configuration

The next step is to choose the cache and memory parameters. The core comes with 32KB cache, 64KB SRAM and 2MB Flash and gives the possibility to turn off caching and configure part of the 32KB of cache as additional RAM. We want to know what is more beneficial for our application – to have more memory or use caching. The situation is depicted in Figure 2. We write two configuration files: C_{cache}, in which we use the cache as such, and C_{ram}, in which we use it as RAM where we allocate the C stack space that usually shows a high degree of reuse. To see how the tasks run on a system with cache, we run TE_{55xx} with inputs E_{55xx} and C_{cache}. To assess the performance without the use of cache, we run TE_{55xx} with inputs E_{55xx} and C_{ram}. Based on the analysis results we can decide with more confidence which of the hardware configurations to choose.

An important question is how the choice of a compiler affects the results. Opposed to some existing simulation techniques that work solely on the source-code level and completely ignore the effect the compiler has on the program's performance, both aiT and TimingExplorer operate on the binary executable and are thus able to achieve high precision. Ideally during the architecture exploration phase the user has the compiler that would be used in the final project at his disposal (for instance because it has been used to build the previous version of the system). Using the actual compiler to obtain an executable is the best way to obtain precise estimates. However when using the production compiler is impossible for some reasons, an executable compiled with a different compiler can be used for the analysis. A preferable solution in this case is to use a compiler that is available for all configurations under consideration. In the examples discussed above we could use gcc for MPC5xx and MPC55xx to obtain E_{5xx} and E_{55xx} respectively. The goal of TimingExplorer is to assess hardware effects on the same code. The observed effects can be expected to also hold when the production compilers are not used provided that the same or at least comparable optimization settings are used.

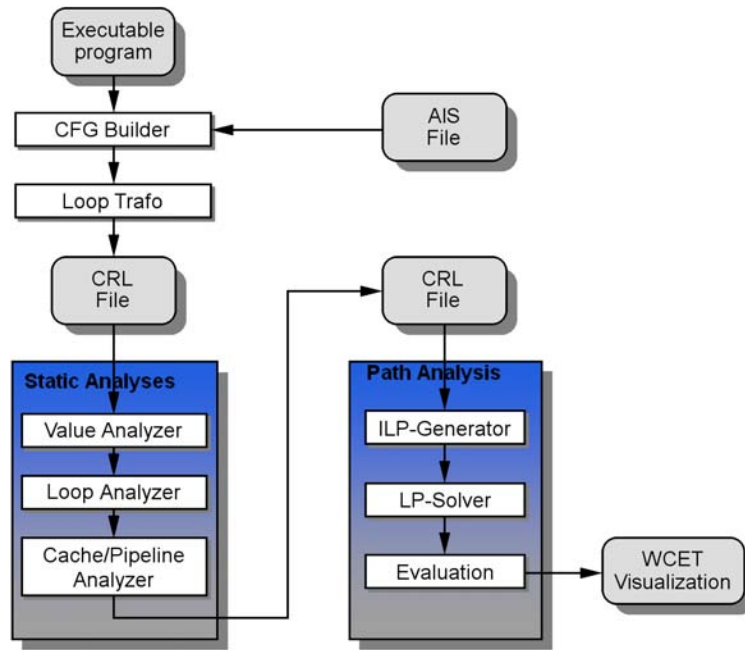


Figure 3. Phases of WCET computation

3. aiT WCET Analyzer

aiT WCET analyzer [7] automatically computes tight upper bounds of worst-case execution times of tasks in real-time systems, a task in this context being a sequentially executed piece of code without parallelism, interrupts, etc. As an input, the tool takes a binary executable together with the start address of the task to be analyzed. In addition, it may be provided with supplementary information in the form of user annotations. Through annotations the user provides information aiT needs to successfully carry out the analysis or to improve precision. Such information may include bounds on loop iterations and recursion depths that cannot be determined automatically, targets of computed calls or branches, description of the memory layout, etc. The result of the analysis is a bound of the WCET of the task as well as a graphical visualization of the call graph in which the WCET path is marked in red.

Due to the use of caches, pipelines and speculation techniques in modern microcontrollers to improve the performance, the execution time of an individual instruction can vary significantly depending on the execution history. Therefore aiT precisely models the cache and pipeline behavior and takes them into account when computing the WCET bound. In our approach [8] the analysis is performed in several consecutive phases (see Figure 3):

1. *CFG Building:* decoding and reconstruction of the control-flow graph from a binary program [20];
2. *Value Analysis:* computation of value ranges for the contents of registers and address ranges for instructions accessing memory;
3. *Loop Bound Analysis:* determination of upper bounds for the number of iterations of simple loops;
4. *Cache Analysis:* classification of memory references as cache misses or hits [6];

5. *Pipeline Analysis:* prediction of the behavior of the program on the processor pipeline [14];
6. *Path Analysis:* determination of a worst-case execution path of the program [21].

aiT uses abstract interpretation [2] to approximate the WCET of basic blocks (steps 2, 4 and 5) and integer linear programming to compute the worst case execution path (step 6).

Furthermore within the INTEREST project [13] aiT has been integrated with two model-based design tools – ASCET [5, 10] and SCADE [4, 9], which are widely used in the automotive and avionic domain respectively. Both tools provide automatic code generators and allow their users to start an aiT WCET analysis from within the graphical user interface, providing them with direct feedback on the impact of their design choices on the performance.

4. From aiT to TimingExplorer

4.1. Overview

aiT is targeted towards validation of real-time systems, so it aims at high precision and closely models the underlying hardware. However in early design phases one might want to investigate the behavior of code on an architecture that closely resembles a certain existing microcontroller for which no high-precision model is (yet) available. Therefore a flexible, yet simple way to specify the memory layout and behavior is necessary. Such a specification should make it possible to run analysis for microcontrollers that are still in development and for which no hardware models exist yet. The way cache and memory properties are specified in TimingExplorer is described in Section 4.2.

Furthermore aiT is sound in the sense that it computes a safe overapproximation of the actual WCET. Depending on the processor this can lead to high analysis times (typically up to several minutes for a task). Such analysis times are acceptable in the verification phase, but are undesirable for configuration exploration. Moreover as opposed to verification for which soundness is of utmost importance, for dimensioning hardware an overapproximation is not always necessary. As long as the tool reflects the task's timing behavior on the architecture and allows comparison of the timing behavior on different configurations, slight underapproximations are acceptable.

Therefore some precision is traded against ease of use, speed and reduced resource needs e.g., through performance optimizations concerning the speed and memory consumption of the pipeline analysis (Section 4.3). Improved usability will be achieved by incorporating automatic source-level analyses that will reduce the need for manual annotation. The types of source-level analyses we are currently working on are described in Section 4.4.

4.2. Cache and Memory Specification

To enable the user to experiment with different configurations in the architecture exploration phase, we have made the cache and memory mapping in TimingExplorer completely parameterizable through annotations. It is possible to set the cache size, line size, replacement policy and associativity independently for the instruction and data cache. Furthermore one can choose how unknown cache accesses are to be treated – as cache hits or cache misses. For example the following specification describes two L1 caches, a 16 KB instruction and a 32 KB data cache that both use an LRU replacement policy:

```
cache instruction
  set-count = 128, assoc = 4, line-size = 32,
  policy = LRU, may = empty
and data
  set-count = 256, assoc = 4, line-size = 32,
  policy = LRU, may = empty;
```

With respect to the memory map, one can specify memory areas and their properties. A memory area is specified as an address range. Useful properties include the time it takes to read and write data to the area, whether write accesses are allowed or the area is read-only, if accesses are cached, etc.

4.3. Pipeline Analysis

To speed up the analysis and reduce memory needs TimingExplorer uses parametric pipeline analysis and computes local rather than global worst-case time as done by aiT. The difference between the latter two kinds of computations is what the tool does when the abstract cache and pipeline state is about to fork into two or more successor states because of imprecise information. This happens for instance when a memory access cannot be classified as cache hit or cache miss. In the case of global worst-case, the tool creates all successor states and follows their further evolution. In contrast, local worst-case means that it immediately decides which successor is likely to be the worst, and only follows the evolution of this single state.

For example, consider that we have two consecutive unclassified memory accesses a_1 and a_2 and let h_i denote the case when a_i is a cache hit and m_i when it is a cache miss. aiT uses global worst-case computation, so after the first unclassified access it creates the superstate $\{(h_1), (m_1)\}$ and after the second – the superstate $\{(h_1, h_2), (h_1, m_2), (m_1, h_2), (m_1, m_2)\}$. From that point on it performs every computation for each of the four concrete states. At the end it compares the resulting times and chooses the highest one. Due to timing anomalies it is not necessary that the worst case is the one resulting from the state (m_1, m_2).

On the other hand TimingExplorer uses local worst-case computation. Unless anything else is specified, this means that a memory access takes longer to execute if it is a cache miss. Therefore after the first unknown access TimingExplorer will simply assume that it is a cache miss and create the state (m_1) and after the second the state (m_1, m_2). It proceeds the execution with a single state (and not with a superstate like aiT).

It should be noted that because of the local worst-case computation the result is not necessarily an overapproximation of the WCET. It is sound in the case of fully timing compositional architectures i.e. architectures without timing anomalies [22], but can be an underestimate otherwise. However the use of local worst case may drastically improve the performance in the cases when there are frequent pipeline state splits.

4.4. Source-code Analyses

4.4.1. Overview

To successfully analyze a task, aiT and TimingExplorer need information about the number of loop iterations, recursion bounds, targets of computed calls and branches. Partially such information can

be automatically discovered when analyzing the binary, e.g., the loop bound analysis mentioned in Section 3 can discover bounds for the number of iterations of simple loops. However information that cannot be discovered automatically has to be supplied to the tool in the form of annotations. To reduce the need of manual annotation, source level analysis will be incorporated that will try to determine auxiliary information that TimingExplorer can extract from the executable.

It should be noted that due to compiler optimizations, the information extracted from the source code does not necessarily correspond to the behavior of the executable. Consider the loop

```
for (int i=0; i<100; i++)
   ...
```

The compiler could discover that the loop will be executed at least once and move the exit condition from the header to the bottom, turning it into a do/while loop. Doing so it reduces the number of jumps by two, which can improve the performance, since jumps often cause a pipeline stall. Furthermore the test condition will be evaluated 100 times (and not 101 times as we will discover by source-level analysis). The resulting overapproximation will result in some precision loss. Whereas in a verification phase the analysis precision is of utmost importance and any loss of precision is undesirable, in an exploration configuration stage the overapproximation is often acceptable because of the time and effort gain from reducing the manual annotation process. Furthermore, the results of the source-level analyses are independent of the compiler and the target architecture and therefore the resulting precision losts will be the same for each of the analyzed configurations. Thus despite the possible overapproximation introduced by incorporating source-level analyses, the comparison of different configurations will still be precise.

Currently, we are working on source level analyses for extracting loop bounds [12] and targets of function calls when function pointers are used [18]. Both of these analyses are implemented with help of the SATIrE framework [17], the LLNL-ROSE compiler [15] and PAG [16].

4.4.2. Function-pointer Analysis

The use of function pointers makes the static analysis of a program hard, because in the general case it cannot be determined which functions will be called at runtime. This leads to an imprecise control flow graph and therefore imprecise results. Unfortunately function pointers are quite common in automotive software and manual annotation should be avoided, because it can be cumbersome and error-prone. Therefore we plan to integrate an automatic analysis for function pointer resolution.

For each function pointer variable in the program the analysis computes a points-to set containing all memory locations whose address might be stored in the variable. The analysis information is stored in a recursive fashion thus making it easy to handle function pointers that are stored within arbitrarily complex objects, like structures with arrays of function pointers. As an output the analysis can create annotations for TimingExplorer, which will relieve the user from determining targets of function pointer calls manually.

During our experiments we were able to successfully analyze several medium-sized C programs. An overview of our test suite is given in Table 1. For the programs in this suite, the analysis detected all function pointers and a manual inspection of the results gave the impression, that they were close to an optimal solution. Our first attempts to analyze real automotive software were not successful,

Program	Version	Lines of code	Indirect calls
diction	0.7	2 037	3
grep	2.0	12 417	3
gzip	1.2.4	8 163	4
sim6890	0.1	3 290	97

Table 1. Test suite for the source-level analysis of function pointers

primarily because of problems in LLNL-ROSE and SATIrE. In some cases this was caused by the use of special language keywords, since the projects had been developed using compilers for target architectures with small word size and used special language constructs. Unfortunately SATIrE also had problems with certain C-constructs (e.g., valid source made it crash).

4.4.3. Loop-bound Analysis

In the case of loop bound analysis we express the upper loop bound as a parametric formula depending on *parameters* – variables and expressions that are constant within the loop. The symbolic formula is constructed once for each loop in an analysis run that precedes the rest of the analyses. Afterwards as soon as the value analysis obtains bounds on the values of all the parameters in a certain formula, a concrete bound on the loop iteration count for that case can be computed with low computational effort.

In an initial evaluation phase we used the Mälardalen WCET benchmark suite [23] to compare our analysis to oRange [1] and the loop bound analysis component in SWEET [3]. The tests showed that:

- The analysis detects and assigns a loop bound to all loops established with C loop constructs except for the cases of loops inside recursive functions or when LLNL-ROSE and SATIrE meet unsupported constructs.
- The precision of the computed loop bounds (assuming that precise values of parameters are provided by an external value analysis) is higher than the one of SWEET and in most cases higher or equal to the precision of oRange.
- The analysis can cope with automatically generated code.
- The analysis is especially effective for typical counter-based loops, which are common in generated code.

Therefore we believe that incorporating source-level loop-bound analysis in TimingExplorer will drastically reduce the number of loops that have to be annotated manually. In the case, when even after the automatic analysis some loop bounds are unknown and the user does not want to spend time on specifying their bounds manually, we have foreseen the possibility to give a global default bound to all loops of unknown iteration count.

5. Integration in a System-level Analysis

After incorporating the source-code level analyses discussed in the previous section in TimingExplorer, we will have a tool capable of predicting the WCET of a task on a selected target architecture

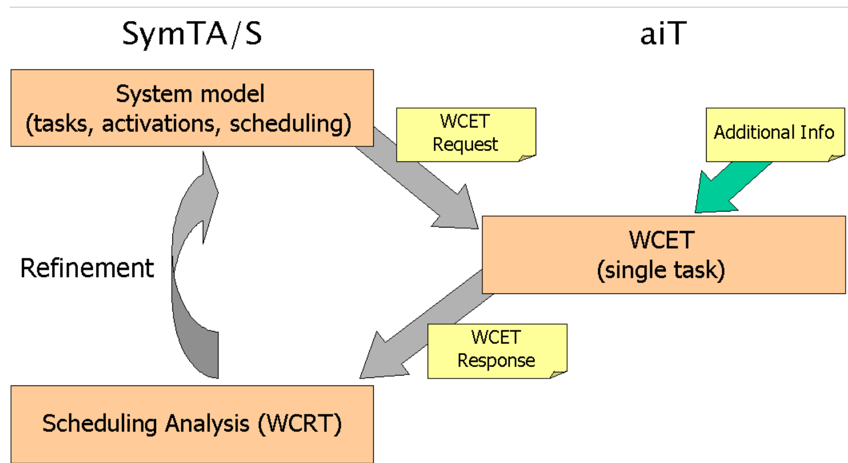


Figure 4. Flow of requests and responses between aiT and SymTA/S

almost automatically. Yet as presented so far, the tool will work only for single tasks without interrupts or exceptions. On the other hand, schedulability analyzers can determine the overall worst-case response time of a system provided the corresponding results for single tasks are known.

Within the European FP6 STREP Project INTEREST [13], a generic information exchange format called XML Timing Cookies (XTC) has been developed. With its help, users of a scheduling analysis tool can cause aiT to perform timing analyses at the code level, whose results are mapped back to the scheduling tool in a fully automatic way. In particular a coupling of aiT with SymTA/S has been realized (Figure 4).

SymTA/S [11, 19] of Symtavision GmbH is a modular tool-suite for scheduling analysis and optimization for electronic controllers, buses/networks and complete embedded real-time systems. It computes the worst-case response times of tasks and the worst-case end-to-end communication delays taking into account the WCETs of the tasks, scheduling, bus arbitration, possible interrupts and their priorities. Two SymTA/S modules that are of special interest for architecture exploration are the *Sensitivity Analysis* and *Design-Space Exploration*. The Sensitivity Analysis module allows its user to determine the robustness of a system and its extensibility for future functions. With the help of the Design-Space Exploration plug-in one can evaluate alternative system configurations and perform system optimization. One specifies which system parameters can be varied and defines optimization objectives, and the tool presents him the most interesting alternatives and the corresponding trade-offs.

We plan to use XTC and integrate TimingExplorer into the SymTA/S system-level architecture exploration analysis. After such integration an overall timing analysis can be performed that will allow users to assess how much room there is for estimation errors and how much flexibility remains for later changes. The combination of code-level and system-level architecture exploration will lead to informed decisions with respect to which architectures are appropriate for an application.

6. Conclusion

We have presented TimingExplorer – a source-level worst-case timing estimation tool to assist in the exploration of alternative system configurations in early design phases. It is an extension of

AbsInt's aiT WCET Analyzer, provides a flexible way to specify an architecture of interest and yields precise estimates of the time it will take a task to run on the modelled architecture. Furthermore TimingExplorer takes all program executions on all inputs into account so that critical corner cases are automatically considered.

We gave an example how a suite of TimingExplorers is to be used to decide what architecture to use for a project and explained the way TimingExplorer differs from aiT. We described two types of source-code analyses that we want to incorporate in TimingExplorer to minimize the need for manual annotation. We also suggested a way to integrate the tool in a system-level exploration framework so that the overall timing performance of the system can be estimated.

We believe that a suite of TimingExplorers will considerably facilitate the process of dimensioning hardware at the beginning of a project. It will allow system architects to assess performance considerations, such as processor capacity, memory layout, cache sizing etc. Thus when choosing what configuration to use they will no longer rely solely on experience and intuition as they do now, but will have an estimate of how the software will behave on the selected architecture at hand.

7. Acknowledgements

This work has partially been supported by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Program.

References

[1] CASSÉ, H., DE MICHIEL, M., BONENFANT, A., AND SAINRAT, P. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proceedings of the 14th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008)* (Kaohsiung, Taiwan, 2008).

[2] COUSOT, P., AND COUSOT, R. Abstract Interpreatation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)* (Los Angeles, California, 1977).

[3] ERMEDAHL, A., SANDBERG, C., GUSTAFSSON, J., BYGDE, S., AND LISPER, B. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis (WCET 2007)* (Pisa, Italy, 2007).

[4] Esterel Technologies. SCADE Suite. `http://esterel-technologies.com/products/scade-suite`.

[5] ETAS Group. ASCET Software Products. `http://www.etas.com/en/products/ascet_software_products.php`.

[6] FERDINAND, C. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[7] FERDINAND, C., AND HECKMANN, R. aiT: Worst-Case Execution Time Prediction by Static Programm Analysis. In *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions*. Toulouse, France, 2004, pp. 377–384.

[8] FERDINAND, C., HECKMANN, R., LANGENBACH, M., MARTIN, F., SCHMIDT, M., THEILING, H., THESING, S., AND WILHELM, R. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the First Workshop on Embedded Software (EMSOFT 2001)* (Tahoe City, CA, USA, 2001), vol. 2211 of *Lecture Notes in Computer Science*, Springer-Verlag.

[9] FERDINAND, C., HECKMANN, R., LE SERGENT, T., LOPES, D., MARTIN, B., FORNARI, X., AND MARTIN, F. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS 2008)* (Toulouse, France, 2008).

[10] FERDINAND, C., HECKMANN, R., WOLFF, H.-J., RENZ, C., GUPTA, M., PARSHIN, O., AND WILHELM, R. Towards integrating model-driven development of hard real-time systems with static program analyzers. In *SAE 2007* (Detroit, USA).

[11] HENIA, R., HAMANN, A., JERSAK, M., RACU, R., RICHTER, K., AND ERNST, R. System level performance analysis – the SymTA/S approach. In *IEEE Proceedings Computers and Digital Techniques* (2005), vol. 152.

[12] HONCHAROVA, O. Static Detection of Parametric Loop Bounds on C Code. Master's thesis, Saarland University, 2009.

[13] INTEREST project. `http://www.interest-strep.eu`.

[14] LANGENBACH, M., THESING, S., AND HECKMANN, R. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium (SAS 2002)* (Madrid, Spain, 2002), vol. 2477 of *Lecture Notes in Computer Science*, Springer-Verlag.

[15] LLNL-ROSE. `http://rosecompiler.de`.

[16] MARTIN, F. *Generation of Program Analyzers*. PhD thesis, Saarland University, 1999.

[17] SCHORDAN, M. Combining tools and languages for static analysis and optimization of high-level abstractions. Tech. rep., TU Vienna, Austria, 2007.

[18] STATTELMANN, S. Function Pointer Analysis for C Programs. Bachelor's thesis, Saarland University, 2008.

[19] Symtavision. SymTA/S. `http://www.symtavision.com/symtas.html`.

[20] THEILING, H. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing and Applications Symposium (RTCSA 2000)* (Cheju Island, South Korea, 2000), IEEE Computer Society.

[21] THEILING, H., AND FERDINAND, C. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)* (Madrid, Spain, 1998).

[22] THIELE, L., AND WILHELM, R. Design for timing predictability. *Real-Time Systems 28*, 2-3 (2004), 157–177.

[23] WCET benchmarks project. `http://www.mrtc.mdh.se/projects/wcet`.