Proceedings of the

7th International Workshop on

Worst-Case Execution Time Analysis

(satellite event to ECRTS'07)

Pisa, Italy

July 3, 2007

`http://www.irit.fr/wcet2007/`

IRIT
CNRS
INPT
UPS
UT1
Institut de Recherche en Informatique de Toulouse

MESSAGE FROM THE WORKSHOP CHAIR

Welcome to the 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007), a satellite event to the 19th Euromicro Conference on Real-Time Systems (ECRTS 2007).

The goal of this workshop is to bring together people from academia, tool vendors and users in industry. Topics of interest include:

- Approaches for WCET computation
- Flow analysis
- 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 and case studies
- Testing Methods for WCET analysis
- Tools for timing analysis
- Design for Timing Predictability
- Integration of WCET analysis into the development process
- Compiler optimizations for worst-case paths
- WCET analysis for multi-processors, multi-cores or SMTs

I would like to thank:

- all the authors and participants who will make this workshop a successful event
- the Steering Committee who gave me valuable advice
- the Program Committee who reviewed papers and were very reactive during the PC email discussions
- the external reviewers who helped in alleviating the task of the Program Committee
- the chairpersons who accepted to lead the discussions within the technical sessions
- Giorgio Buttazzo who was responsible for the ECRTS local arrangements

I hope that you will enjoy the workshop!

Christine Rochange
Institut de Recherche en Informatique de Toulouse, France

WCET'07 STEERING COMMITTEE

Guillem Bernat, University of York, UK
Jan Gustafsson, Mälardalen University, Sweden
Peter Puschner, Technische Universität Wien, Austria

WCET'07 PROGRAM COMMITTEE

Antoine Colin, Rapita Systems Ltd., UK
Raimund Kirner, Technische Universität Wien, Austria
Stefan Petters, National ICT Australia Ltd., Australia
Christer Sandberg, Mälardalen University, Sweden
Stephan Thesing, Rücker GmbH, Germany

WCET'07 EXTERNAL REVIEWERS

Guillem Bernat, Adam Betts, Stefan Bygde, Hugues Cassé, Andreas Ermedahl, Christian Ferdinand, Jan Gustafsson, Susanne Kandl, Martin Kirner, Nicholas Merriam, Markus Pister, Isabelle Puaut, Peter Puschner, Bernhard Rieder, Marcelo Santos, Marc Schlickling, Martin Schöberl, Ingomar Wenzel, Patryk Zadarnowski, Michael Zolda

WORKSHOP PROGRAM

Measurements or Static Analysis or Both?

Stefan M. Petters Patryk Zadarnowski Gernot Heiser*
NICTA† and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

To date, measurement-based WCET analysis and static analysis have largely been seen as being at odds with each other. We argue that instead they should be considered *complementary*, and that the combination of both represents a promising approach that provides benefits over either individual approach. In this paper we discuss in some detail how we aim to improve on our probabilistic measurement-based technique by adding static cache analysis. Specifically we are planning to make use of recent advances within the functional languages research community. The objective of this paper is not to present finished or almost finished work. Instead we hope to trigger discussion and solicit feedback from the community in order to avoid pitfalls experienced by others and to help focus our research.

1 Introduction

Embedded systems are becoming more pervasive by the day, and many of these embedded systems are subject to critical temporal requirements. While many of these systems may not be life critical, missing deadlines may nevertheless be a costly excersise if experienced as degraded functionality or quality of service by millions of end users.

The analysis of worst-case execution times (WCET) is a fundamental building block of any form of real-time analysis. Most of the work to date has been based either on static analysis or on measurements. The research community has predominantly focussed on static analysis, but measurement-based techniques have gained increased significance over the last ten years.

These two principal approaches have largely been seen as mutually exclusive, and proponents of either approach tend to be quite critical of the other. Common concerns voiced about measurement-based analysis are that:

1. it is unsafe, as there are no guarantees that the worst case has been observed and
2. measurements are too expensive if sufficient coverage is to be achieved.

On the other hand, critics of the static-analysis approaches claim that static analysis:

1. is unsafe, as modern architectures are highly complex and thus modelling them is an error prone process, not least due to lack of documentation,
2. raises substantial challenges in terms of portability, and
3. does not support the more creative features used to improve performance in today's architectures.

We believe that ultimately a combination of the two paradigms is required to overcome the issues in both. Specifically, we propose to use measurements to obtain realistic, accurate results and static analysis to back the findings of the measurement phase by establishing that major contributors to the variability of the execution time have been adequately covered. Besides variations in program path, which are usually covered in the computation phase of WCET analysis approaches, caches contribute most substantially to variations in the execution times of software. Establishing whether all cache misses as predicted by static analysis have been observed in the measurements is of substantial help to ensure confidence in results obtained by measurements. Focussing on caches allows for easy verification that the model used is actually correct and provides a high degree of portability of the analysis.

Furthermore, the results of the static analysis of caching behaviour can be used to reduce the over-

*Also with Open Kernel Labs

†National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

estimation produced when analysing the measurements of small untis independently and when conservatively covering any possible dependency between the units.

2 Related Work

The integration of cache analysis into WCET analysis was pioneered by Mueller [1] and Lim et al. [2]. The latter represented a holistic WCET and schedulability analysis that was subject to considerable complexity and was eventually abandoned as a line of research. Mueller's work has been refined over the years [3]. However, the main drawback of the approach is the loss of information inherent to the abstraction processs., Specifically information is lost when the predefined *join* function is used to merge abstract cache states at points in the program where two control flows of a program merge (e.g. after an `if-then-else` construct. Further, abstract analysis, the tool of choice for static program analysers, has proven notoriously resilient to all non-trivial attempts to apply it to programs that manipulate dynamic data structures such as linked lists or those in which pointers to functions cannot be resolved statically. While it can be argued that both features are rarely found in real-time programs, they are nevertheless common in certain critical parts of the system such as dynamic schedulers and page tables.

Ferdinand and Wilhelm [4] have extended Mueller's work by introducing the *must and may* analysis, effectively reducing the amount of information lost by the *join* function and proving that the resulting abstract domain is optimal. Nevertheless, even with these improvements, the analysis still loses some information at junctions of control-flow paths introduced by any chosen program representation. Further, the *must and may* analysis suffers from the same limitations as the earlier approaches when applied to code manipulating dynamic data structures.

Attacking the WCET problem from a different angle, Kirner et al. [5] deployed static analysis to identify a set of input data, which would enforce any possible path combination to be executed, effectively doing a full path enumeration. This set is then fed into the program and measured on real hardware. In order to manage complexity, the program under test is divided into program segments which are tested and measured independently. The approach did not support caches and thus is not applicable to our work.

Yamamoto et al. [6] approached the problem of ensuring measurement coverage of cache states by measuring each basic block in isolation in a best case scenario; in other words, all referenced memory locations are preloaded into the caches. A separate cache analysis provides a worst-case cache-miss scenario for the given basic block and enables the addition of the *cost* of these cache misses in the computation stage of the analysis process. The exact cache simulator used is not described in their paper, however, the analysed programs in their evaluation are sufficiently small to allow a brute force computation of the cache states.

3 Potoroo

A brief introduction to the overall framework is necessary to set the proposed approach into context. The Potoroo project aims to analyse the kernel primitives of the L4 microkernel API [7] for their WCET to enable real-time systems to be built ontop of the kernel. So far, we have developed a toolset which allows the measurement-based analysis of the kernel. In terms of the general approach it follows the paradigm used in [8].

The executable code of the program under test is analysed to extract the control-flow graph (CFG). By using the executable code, all compiler optimisations and preprocessor modifications are considered. The analysis tries to be minimal by focussing mainly on control-flow changing instructions. However, this implies that register-indirect branches are particularly hard to resolve. Instead of a full analysis of the code, we have chosen to use a source code parser developed in the Goanna project at NICTA [9] and use debugging information in the executable to find corresponding parts in the source code.

Traces may be generated either by software instrumentation, hardware support, or using cycle-accurate simulators. Software instrumentation is subject to overhead and may quickly become too high a burden in a running system. Cycle-accurate simulators, however, raise the question of accuracy of the model in the simulator — "mostly right" is not good enough. Hardware supported tracing usually makes use of debugging ports implemented on the processor die, like the ETM macrocell in some ARM processors. Traces are taken usually on a basic block level, where the time stamp of the first in-

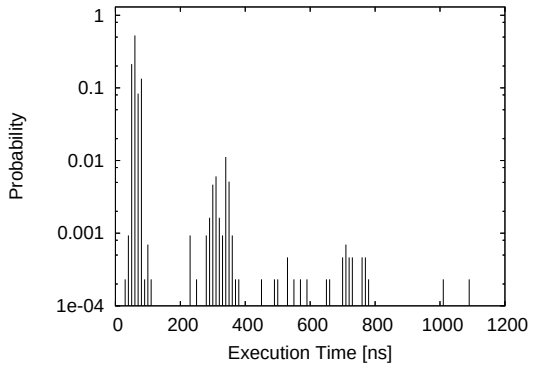


Figure 1: Sample ETP

struction of each basic block is stored alongside a basic block identifier. All the execution time measurements for a basic block are interpreted as a probability distribution called execution time profile (ETP). While basic blocks exhibit their WCET easily compared to entire programs, there are no guarantees that a given block has been completely represented in the ETP.

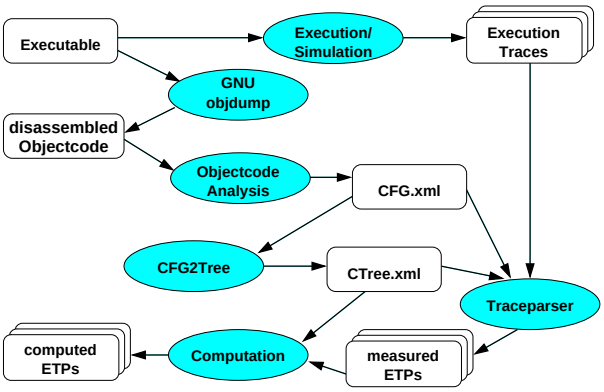


Figure 2: Toolset Overview

The traces are translated into ETPs, an example of which is depicted in Figure 1. This is performed using the control-flow graph previously established. Besides the work of actually producing the traces, this is the most computationally-expensive part of the approach.

The CFG is also translated into a tree, which directs the combination of ETPs to form ETPs describing larger code constructs. Using the tree ensures that any possible path combinations are considered.

For this paper the combination of sequential code constructs is of particular relevance. The toolset employs the supremal convolution [10, 11] for this. The supremal convolution combines two distributions in such a way that any possible dependency between the two distributions is conservatively covered in the result, thus ensuring a safe combination of two ETPs. However, a major drawback of supremal convolutions is that they are very conservative and tend towards a yes/no decision instead of a profile when many ETPs are combined [12].

4 Basic Idea

In the previous section we have identified two fundamental challenges to the approach we are taking in analysing the kernel.

1. Ensuring sufficient test coverage on basic block level.
2. Avoiding the overly-conservative nature of the supremal convolution without jeopardising safety.

Looking at the variability of the execution time in Figure 1 we can see that the ETP is clustered. These clusters can be attributed to cache misses, which are dominating the execution time of a given piece of code. Guaranteeing that the code has actually experienced its worst case of cache misses during the execution would go a long way to guaranteeing sufficient measurement coverage.

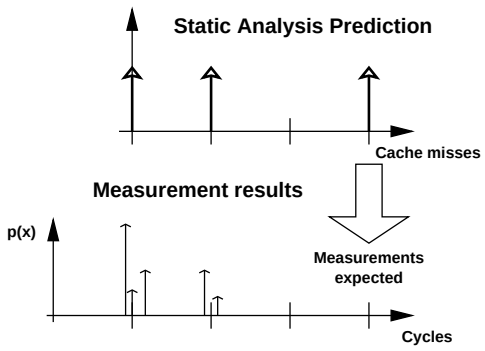


Figure 3: Coverage

In order to tackle this, we aim to establish for each ETP the different cache-miss scenarios expected and compare that to the measured ETP as depicted in Figure 3. While the creation of a complete and accurate model of a system including processor core, caches and peripherial devices is non-trivial and raises the issue of portability, caches themselves are only subject to a few parameters which can be easily established and verified for a given system [13]. In order to be able to make the connection between cache misses predicted and the measured

ETP, it is necessary to reason about the cache-miss penalty actually imposed on a given cache miss.

While caches are used to mitigate the effect of long memory access latencies, modern processors try in various ways to mitigate the effect of cache-miss penalties. Critical-word-first loads by caches avoids the overhead of loading data which is not immediatly required, if the request does not hit the first word in a cache line. Out-of-order execution enables the program to progress on instructions which are not dependent on the memory location being loaded. A side effect of out-of-order execution is that instructions independent of the cache miss are executed. Thus a cache miss at a given point in the program reduces the entropy of states the CPU may be in during the execution of subsequent instructions after the data has been fetched from memory.

Load/store architectures tend to tag registers waiting for outstanding memory requests, to enable continued execution until the register is actually used. This enables a smart compiler to make use of instruction scheduling to preload registers as early as possible to avoid as much of the maximum cache miss penalty as possible. Contrary to out-of-order execution, the pipeline is usually drained of instructions preceding the cache-miss causing instruction. Some architectures such as the ARM9EJ-S processor core allow for only a single outstanding memory transaction. However, other processors such as the XScale processor family allow for several outstanding requests, by implementing fill buffers and pend buffers.

Applying the above discussion to the environment we are performing our analysis in, we make the following observations:

1. The ARM9EJ-S is only subject to a single outstanding memory transaction, forcing a stall on subsequent loads.
2. The ARM-gcc compiler typically uses loaded registers within three instructions thereby making little or no use of the reduced penalty of a delayed load.

Any approach performing coverage analysis should inherently have information about dependencies between the cache misses of subsequent basic blocks (and possibly even beyond that). Exploiting these dependencies as depicted in Figure 4 allows, on the one hand, more realistic bounding of ETPs, and on the other hand, the reduction of the overall WCET.

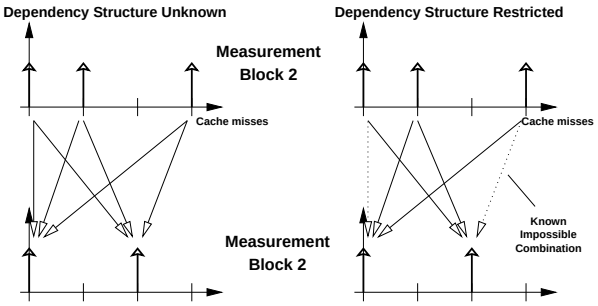


Figure 4: Dependency Analysis

5 Static Analysis Approach

Static analysis is well-established as a powerful tool for computing the WCET of a program. In particular, abstract interpretation, the tool of choice for static program analysers, is an attractive technique for WCET analysis, as it provides a method for a formally-provable derivation of concrete program properties such as cache misses or even actual bounds on the execution time. Unfortunately, in the past, all applications of static analysis in the area have been hampered by the limitations, described in section 2, inherent to the abstract interpretation technique.

5.1 Motivation

In our work, we observe that the problem of deriving the WCET of a given program using static analysis can be viewed as a search for a proof of a desired program property. In particular, abstract interpretation can be viewed as a way of deriving a constructive proof of the desired property by computing that property directly from the structure of the program. However, if we knew the property in the first place (for example, through empirical measurement of program's behaviour) we could, in principle, construct an indirect proof of the same result. In particular, we can attempt to prove the result by showing that no possible execution scenario can result in an answer different from the assumed one. Conversely, we can disprove our hypothesis by searching for a suitable counter-example during program analysis. In the remainder of this section, we argue that the indirect approach is particularly well suited to the problem of computing the number of cache misses experienced during execution of a program.

5.2 The Basic Approach

We take the set of cache miss counts observed during measurement of the program as a hypothesis, which we subsequently attempt to prove or disprove through static analysis of the program. The problem is simpler than attempting to compute the cache behaviour "from scratch" since the measured answer provides finite bounds on the amount of computation performed during analysis, independent of the bounds imposed by the particular abstract domain and the associated join function. This gives us more leeway in the design of the abstract domain, and in fact permits us to perform the static analysis of the program with virtually no loss of information at all. In particular, observe that:

1. Since the measured set of cache miss counts is finite, it can always be obtained after a finite number of steps during abstract analysis, provided that we take some simple precautions in the design of our algorithm to avoid divergent chains of computation.
2. If due care is taken during design of the analysis algorithm, the above observation is sufficient to guarantee that the analysis is performed in a "reasonable" amount of time. However, this does not prevent us from taking additional measures to avoid the exponential complexity of complete control path enumeration by combining analysis for sections of the program that are common to two or more potential execution paths. In our approach, we will avoid exponential complexity by binding execution time of our analysis to the number of cache miss counts observed during measurement.

In other words, we can safely "run" the analyser until it has either constrained the set of cache miss counts to a subset of the measured one, or else until it has detected a counter-example to our hypothesis. In the later case, the state of the analyser at the time when the counter-example has been detected provides invaluable clues permitting the user to extend the measurement suite to cover the omitted execution scenarios.

Note that this approach is strictly limited to analysing those programs for which a "perfect" measurement suite can actually be constructed from a finite number of test cases. This excludes, among others, non-terminating programs. Fortunately, this is precisely the class of programs suitable for use in real-time applications and accordingly, covers all programs that we are concerned with.

The remaining subsection outline our implementation of this technique.

5.3 Source Program Preparation

First, we translate the input binary program into a purely-functional representation using the technique pioneered by Chakravarty, *et al.* [14]. We choose a normal form of the continuation-passing style of lambda calculus as our program representation for its similarity to the low-level treatment of control flow on typical processor architectures. In the purely-functional form, all basic blocks are translated into functions with loops represented by recursion. Further, all global variables are replaced by additional function arguments "threaded" throughout the control-flow path of the program. This step is necessary for pragmatic reasons, since the subsequent program transformations would become prohibitively-expensive without the detailed data-flow information explicit in purely-functional programs.

Note that, in this paper, we use the term "function" in the declarative programming sense of the word, rather than to refer to the procedures of the input program. Every function in our analysis corresponds loosely to a basic block of the input program.

5.4 Cache Analysis

Next, we transform the input program into a new *analyser program* that dynamically computes the cache miss counts of the original program. This step is very similar to a conventional abstract analysis, and uses the same form of an abstract domain to represent the cache miss counts. However, since the solution we seek is computed dynamically during execution of the analyser program rather than statically in the course of analysis, we never have to join abstract values in the analyser program as described in section 2. During any given actual execution of the analyser, only one of all possible control flow paths can be followed. In other words, the analyser program provides a compact, finite representation of the large (and potentially infinite) number of all possible control flow paths that would have to be followed to obtain precise cache miss counts for every possible execution scenario, just like the original program provided a compact finite representation of all control flow paths of the original program. Note that the resulting program encodes

the *precise* cache miss count for every possible control flow path without any loss of information. Also note that such translation of an input program into an analyser program is performed implicitly by every abstract analysis algorithm, although the resulting program is rarely "materialised" into an actual data structure, and typically remains encoded implicitly in the state of the static analyser. Further, during conventional abstract analysis, the control structure of the translated program is simplified at the expense of precision to ensure termination of the analyser.

Besides this translation, we also perform a number of standard optimising transformations of the generated programs, including constant folding, copy propagation and dead code elimination. Since the typical calculations involved in computing cache miss counts are relatively straight-forward in comparison to the work done by the original input program, we expect these simple transformations to result in a dramatic reduction to the size of the analyser program. In particular, large chunks of code that do not affect cache behaviour should disappear from the program, thus substantially reducing the cost of the subsequent stages of the whole process. We also perform an induction variable analysis to reduce many common loop patterns such as those used to obtain a sum of an arithmetic series into a simple scalar expression.

5.5 Coverage Analysis

Finally, we analyse the transformed program to verify that it can only return values from the set of cache miss counts observed during measurement. In other words, we seek to find the maximal set of input arguments for which the analyser program returns an answer within the range specified by the measured set. Our solution first constructs an inverse of each function f in the program (in other words, a function f^{-1} such that $f^{-1}(x) = \mathbf{Y}$ iff, for all y in the set $\mathbf{Y}$, $f(y) = x$.) It is a remarkable fact that such inversion can be performed relatively easily for all functions that may be encountered in an analyser program. This is because we are seeking total inversion only, rather than partial inversion (where some of the input arguments to the original functions remain fixed) which is a harder problem. While the ranges of the inverted functions grow quickly with the number of original function parameters, as will be shown shortly, this is not a problem in our application because the size of those ranges is used to bind the depth of our analysis and facilitate early termination of our algorithm. The true exponential growth is therefore never reached and the overall complexity of the algorithm is a logistic function of the size of the measured set and the number of basic blocks in the input program. The term *logistic function* relates to an intial exponential growth of the function which subsequently slows and finally stops.

The analysis maintains a work list of inverted functions annotated with a set of their input arguments. Initially, the work list contains only those functions that correspond to the leaves of the call graph of the original program, each annotated with the set of cache miss counts obtained through measurement. The algorithm proceeds by extracting each item from the work list in turn, and terminates when the work list becomes empty.

A single work list entry is analysed by applying the given input value set to the corresponding inverted function, thus obtaining the maximal set of corresponding program inputs. We recognise two scenarios:

1. If the resulting set of values is unconstrained (as will often happen by the nature of function inversion), we have determined that the measurements have been exhaustive along the corresponding control-flow path in the original program. Accordingly, the function is removed from the work list.
2. Otherwise, we determine the set of callers of the function under consideration in the original (non-inverted) program, and add the corresponding inverted functions to the work list. If no such functions exist, we have just examined the entry block of the original program, and accordingly report the resulting set of constraints to the user.

All constraints reported to the user as a result of the second point above represent constraints on the input of the original program that must be satisfied in order for the program's cache behaviour to remain within the measured set of cache miss counts. Accordingly, all input values outside of the constraint set represent counter-examples to our hypothesis.

5.6 Algorithmic Complexity

In the worst case, the algorithm may analyse all individual control-flow paths through the program. However, in practice the worst case is incredibly difficult to achieve, as the execution of our algorithm is bounded by the size of the constraint set,

which itself grows exponentially during the function inversion process. This means that the actual complexity of the program is a logistic, rather than an exponential function. In fact, we believe that the amortised complexity of our algorithm for all terminating input programs is polynomial (quadratic) in the number of functions (basic blocks) in the program. More research is needed to substantiate this result.

6 Conclusions

In this paper we have outlined our approach to supporting probabilistic measurement-based WCET analysis with static analysis. The static analysis is based on a functional representation of the code investigated and an abstract interpretation of representation. The goal is to establish sufficient measurement coverage, and to reduce overestimation of conservative combination of ETPs by conservatively covering any possible dependencies between them. Future work will largely center on finishing the implementation of the plan presented and performing the subsequent evaluation.

References

[1] F. Mueller, *Static Cache Simulation and its Applications*. PhD thesis, Department of Computer Science, Florida State University, Tallahassee, FL, USA, July 1994.

[2] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim, "An accurate worst-case timing analysis for risc processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 7, pp. 593–604, 1995.

[3] K. Patil, K. Seth, and F. Mueller, "Compositional static instruction cache simulation," in *Proceedings of the Conference on Language, Compiler and Tool Support for Embedded Systems2004*, (Washington, DC, USA), June11–13 2004.

[4] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Journal of Real–Time Systems*, vol. 17, pp. 131–181, 1999.

[5] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*, vol. 2, UBooks Verlag, Dec. 2005.

[6] K. Yamamoto, Y. Ishikawa, and T. Matsui, "Portable execution time analysis method," in *Proceedings of the 12th International Conference on Embedded and Real-Time Computing and Applications*, (Sydney, Australia), Aug. 2006.

[7] C. van Schaik and G. Heiser, "High-performance microkernels and virtualisation on ARM and segmented architectures," in *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, (Sydney, Australia), NICTA, Jan. 2007.

[8] G. Bernat, A. Colin, and S. M. Petters, "pWCET: a tool for probabilistic worst case execution time analysis of real–time systems," technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[9] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Goanna — A Static Model Checker," in *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems*, (Bonn, Germany), pp. 297–300, Aug. 2006.

[10] R. C. Williamson, *Probabilistic Arithmetic*. PhD thesis, Department of Electrical Engineering, University of Queensland, Brisbane, Australia, Aug. 1989.

[11] G. Bernat, M. Newby, and A. Burns, "Probabilistic timing analysis: An approach using copulas," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 179–194, 2005.

[12] S. M. Petters, "Execution-time profiles," tech. rep., NICTA, NICTA, Sydney 2052, Australia, Jan. 2007.

[13] T. John and R. Baumgartl, "Exact cache characterization by experimental parameter extraction," in *Proceedings of the 15th International Conference on Real-Time and Network Systems RTNS07*, (Nancy, France), pp. 65–74, Mar. 2007.

[14] M. M. Chakravarty, G. Keller, and P. Zadarnowski, "A functional perspective on ssa optimisation algorithms," in *Electronic Notes in Theoretical Computer Science* (J. Knoop and W. Zimmermann, eds.), vol. 82, Elsevier, 2004.

Automatic Amortised Worst-Case Execution Time Analysis

Christoph A. Herrmann* Armelle Bonenfant† Kevin Hammond*
Steffen Jost* Hans-Wolfgang Loidl‡ Robert Pointon§

Abstract

Our research focuses on formally bounded WCET analysis, where we aim to provide absolute guarantees on execution time bounds. In this paper, we describe how *amortisation* can be used to improve the quality of the results that are obtained from a fully-automatic and formally guaranteed WCET analysis, by delivering analysis results that are parameterised on specific input patterns and which take account of relations between these patterns. We have implemented our approach to give a tool that is capable of predicting execution costs for a typical embedded system development platform, a Renesas board with a Renesas M32C/85U processor. We show that not only is the amortised approach applicable in theory, but that it can be applied automatically to yield good WCET results.

1 Introduction

Worst-case execution time (WCET) analysis is required for a variety of embedded systems applications, especially those with safety- or mission-critical aspects. Common examples include avionics software and autonomous vehicle control systems [14]. Our work aims to construct *fully automatic* source-level static WCET analyses, that are correlated to actual execution costs. Since we must provide formal, automatically-produced *guarantees* on WCET bounds, we base our work on a high-quality abstract interpretation approach (AbsInt GmbH's **aiT** tool [5]), to give low-level timing information for bytecode instructions. We combine this with an equally formal, *type-based* approach that lifts this information to higher-level language constructs so that it can be applied to source programs. The problem is to maintain the strong WCET guarantees we need, while giving good quality information. In this paper, we consider a new approach to constructing WCET analyses, based on the idea of *amortisation* [16]. This represents the first attempt of which we are aware to provide an automatic amortised WCET analysis. We have produced a prototype implementation using our approach, and we report here on some preliminary results obtained using this analysis tool.

*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX.
email: {ch,kh,jost}@cs.st-and.ac.uk.
†IRIT, Université Paul Sabatier, Toulouse, France.
email: bonenfant@irit.fr
‡Ludwig-Maximilians Universität, München.
email: hwloidl@informatik.uni-muenchen.de
§Heriot-Watt University, Edinburgh.
email: rpointon@macs.hw.ac.uk

2 Amortised Time Analysis

Amortised cost approaches [3] allow costs to be averaged according to use. The basic intuition is that by amortising over the time costs incurred by common usage patterns (e.g. that for a stack, every *pop* is balanced by a *push*), we can construct timings that reflect more accurately real worst-case times. Typically, amortised analysis is performed by hand to determine the complexity of programs that involve complex data structures [13]. We have previously, however, applied the approach to give automatically derived, and provably correct, upper bounds on space costs for heap allocations [10]. In both cases, since alternative program execution paths may have very different costs, by amortising over common patterns we can avoid the needless over-estimation that would otherwise occur.

In this paper, we consider how the same approach can be applied to WCET. Our thesis is that such an approach can potentially reduce over-estimation *without losing formal guarantees that the analysis yields a genuine WCET.* We will show below that our amortisation-based WCET analysis can obtain WCET bounds that are close to the actual execution time. We first consider a simple example to illustrate the principles of our approach.

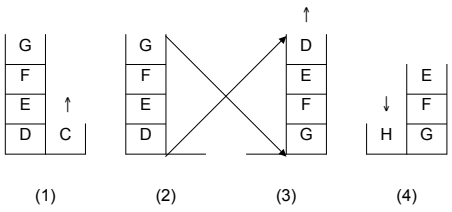


Figure 1: Queue implemented by two stacks

2.1 Example: Implementation of a Queue by Two Stacks

An example that is often used when teaching data structure abstraction and encapsulation is the implementation of a queue as two stacks. We will use this here to illustrate our amortised approach. Figure 1 shows a sequence of four queue configurations, in each of which the left stack is used to enqueue items and the right one to dequeue them. A queue abstraction can be built directly in terms of *push* and *pop* operations on the underlying stacks without needing to break the stack abstraction.

Enqueuing to a stack and dequeuing from a non-empty stack takes constant time. When the dequeuing stack becomes empty after popping C in (1), the contents of the enqueuing stack in (2) are popped in turn, and pushed onto the dequeuing stack as shown in (3). This then allows element D to be dequeued. Finally, new elements can be enqueued as shown in (4).

Reversing the stack by elementwise copying ($(2) \rightarrow (3)$), however, takes time proportional to the number of items on the stack.

There are thus two worst cases. Firstly, the stack could become as large as the total number of elements, n, if the final element is enqueued before the first one is dequeued. The cost for the reversal would then be proportional to n. Secondly, each dequeued element could require a stack reversal if it is dequeued before the next one is enqueued. This would mean n stack reversals in total. Combining both worst cases would yield a worst-case estimation proportional to n^2. *However, both cases cannot occur at the same time*: as there are more stack reversals, the sizes of the stacks to be reversed decrease in size.

Amortised analysis treats these two extreme worst-cases quantitatively: we distribute the cost for the stack reversal equally among all the elements. The costs incurred by each element are: (1) a push for enqueuing; in the stack reversal (2) a test and (3) a pop followed by (4) a push; and for dequeuing (5) a test and (6) a pop. To process all elements, it follows that $6n$ basic stack operations are required.

By considering the queue structure as a whole rather than the operations of its components (stacks) in isolation, we have been able to amortise the worst-case cost, and *reduce it from a quadratic cost to a linear cost*. We can exploit this in an automatic analysis by using a model in which a *potential* [16] is given to each kind of data element. This potential represents the amortised worst-case cost per element of the structure. In the model (but not, of course, in an actual execution!), one unit of potential is released for each operation on an element. The potential must be sufficient to cover all operations. In our example, if we take into account only the stack operations, the potential would be 6 for each element.

2.2 Type-Based WCET Analysis

Our approach [11] is to analyse *source-level* constructs by building on standard type-checking algorithms. By doing this, we are able to gain information about programming constructs that may be lost through a compilation process, and so to obtain a better quality of analysis. Types also allow us to construct a *compositional* analysis, where functions/expressions/modules can be analysed independently. The flip-side is that we must have the program source available at analysis time. However, because we have a compositional analysis, previously obtained analysis information can be attached in the form of *meta-data* to binary programs or library code, and this will reveal nothing about the source implementation, apart from its WCET. In order to analyse source functions that use this code, it is not necessary to be able to *read* the binary file, or even to possess it, but only to see the meta-data.

Our work is undertaken in the context of the domain-specific programming language Hume, which embeds purely functional expressions in a powerful automaton-based process notation [9, 8]. We have constructed formally-correct type-based automatic analyses for determining worst-

case execution-time costs, based on the amortised cost approach described above [11], and implemented our rules in a prototype implementation (available from http://www.embounded.org). We will show some results that can be obtained from our analysis below.

The analysis works as follows: each construct in the source program is given a type using a normal type-inference algorithm. At the same time, the usage of potential is calculated for that expression. (Internal) cost variables are automatically associated with each (sub-)expression, and the analysis will generate a set of constraints over those variables that give an upper bound on the WCET. The constraint set is solved using a linear equation solver, and concrete cost solutions are mapped back to the source program. In this way, WCET costs are associated with each expression and each function that is used in the program.

Note that, while Hume deliberately simplifies the problem of constructing cost models and analyses, in order to allow us to focus on key research questions, rather than worrying about specific complexities, of e.g. C programming, the methods we are developing are, in fact, generally applicable. Hofmann and Jost have shown, for example, how a formally bounded heap analysis could be applied in an object-oriented setting such as Java [10]. Note also that, although our focus is on formally guaranteed WCET, and we have therefore built on abstract interpretation and types, amortisation can also, in principle, apply to other WCET approaches such as probabilistic approaches [1].

3 Obtaining WCET Bounds for HAM Instructions

The problem now is one of obtaining reliable WCET information for simple expressions so that costs for complex expressions can be constructed from the type-based analysis. We do this by first introducing an abstract machine (the Hume Abstract Machine or HAM [6]). We may then systematically determine WCET costs for each HAM instruction. Having determined the cost of each HAM instruction for a given architecture, and knowing how the Hume compiler will translate expressions to HAM instructions, we are able to analyse Hume programs for that architecture without needing to perform any compilation (even to HAM code), or any further analysis on the target machine. This yields a flexible and highly portable analysis.

The approach can yield an acceptable upper bound on WCET: we have shown in a previous paper [2] that composing costs of individual HAM instructions delivers a WCET result that can be within 2% of the cost of the WCET for a sequence of HAM instructions for the Renesas M32C/85U [4] architecture that we will also use in this paper. Processors such as the M32C, which have predictable time behaviour and low power consumption are especially of interest for use in embedded systems such as automotive applications. The processor therefore represents an important class of processor architectures that is employed in safety- and mission-critical systems.

3.1 Low-Level WCET Analysis

In this paper, we distinguish between *measured WCETs*, that are obtained in the obvious way, and *guaranteed WCETs*, that provide a formally guaranteed upper bound on WCET. Probabilistic approaches [1] extend the basic measurement approach by extrapolating from a set of measured WCETs to provide a probability function for the WCET.

Although it can be relatively simple to obtain time information by measurement, even for complex architectures such as a Pentium IV, this can be time-consuming, and it is not always straightforward to determine that the actual worst-case has been covered by the test input. Conversely, for guaranteed WCET obtained by static analysis, it is necessary to formalise the behaviour of the target processor. It can clearly be costly to construct such a model, especially where cache and pipeline effects are involved. For unpredictable architectures, such as the Pentium IV, even if it is feasible to model such a complex system, it may not be possible to construct a model that gives a tight WCET: the WCET could, in some cases, be one or two orders of magnitude greater than the typical execution time. However, once a model and analysis has been constructed, it can be applied repeatedly, and usually without significant programmer effort. A more detailed comparison of WCET approaches up to early 2007 can be found in the paper by Wilhelm et al. [17].

3.2 WCET Information for Individual HAM instructions

Our type-based approach can exploit information obtained using either measured or guaranteed WCETs for single HAM instructions. However, we can only formally *guarantee* upper bound times for source programs, if we use a correspondingly guaranteed approach to obtaining low-level WCET information. In this paper, we investigate both measured and guaranteed WCETs of HAM instructions for the Renesas M32C/85U, using machine code generated from HAM instructions by the IAR C compiler [15].

Guaranteed WCETs for each HAM instruction have been obtained using AbsInt GmbH's **aiT** tool [7]. This gives the WCET for each machine code fragment that is generated for a HAM instruction, using a static analysis approach that computes safe approximations for all possible cache and pipeline states that can occur at any given point in the program.

Measured WCETs for each HAM instruction have been obtained by repeated measurement of the instruction execution time, taking the worst case from 10000 runs, and using the cycle-accurate timer on the M32C to obtain software timings. To ensure accuracy, we use a "two-loop benchmarking" approach, which measures the time required for auxiliary measurement operations separately and subtracts this time from the measurement of interest. In this way, we avoid including measurement costs in the worst-case measurement. In order to guarantee that we measure the WCET for the program, we must, of course, provide an input which incurs the WCET during normal fault-free operation.

4 Example: Drilling Robot

We will now consider a slightly more in-depth example, to show how our amortised approach can be exploited to give WCET results and compare the outcome of the analysis with measured WCET times on the M32C processor.

4.1 Problem description

Our case study here is the simulation of a robot for drilling printed circuit boards. The robot can move a drilling head in fixed-sized increments (here represented as integers) in two dimensions.

```
pos_ok (xpos, ypos) = if xpos==0 && ypos==0
                      then 1 else 0;

step (xpos,ypos,actions,dps) =
 case actions of
    []     -> (dps, pos_ok (xpos, ypos))
  | (A:as) -> step (xpos,  ypos,   as,
                          ((xpos,ypos):dps))
  | (L:as) -> step (xpos-1,ypos,   as, dps)
  | (R:as) -> step (xpos+1,ypos,   as, dps)
  | (U:as) -> step (xpos,  ypos-1, as, dps)
  | (D:as) -> step (xpos,  ypos+1, as, dps);
```

Figure 2: Hume code for Drilling Robot

At each position, the drilling head can perform a drilling action. After all holes have been drilled, the drilling head is to be moved to its starting point. A similar application example would be a camera that can be turned around two axes and can take photo shots at particular orientations.

The robot can perform five operations: `A` is the drilling action; the other actions move the head by one position: `L` leftwards, `R` rightwards, `U` up and `D` down. Operations are described by the user-defined data type `OP` in Hume:

```
data OP = A | L | R | U | D
```

For example, if the action list is the four-element list `R:(D:(A:(L:(U:[]))))` (where `[]` represents the empty list and `(U:[])` constructs the one element list with `U` at the start of the list and an empty remainder), and the robot starts at position `(0,0)`, it will move right to `(1,0)`, down to `(1,1)`, drill, move left to `(0,1)` and finally up to `(0,0)`, the starting position.

The Hume function `step` (Figure 2) simulates each operation carried out by the drilling robot*. `step` takes four arguments: the current X- and Y-positions of the (`xpos` and `ypos`); the list of remaining actions to perform (`actions`); and an accumulating list that records the positions at which a drilling action has happened (`dps`). We perform case discrimation on the list of actions, where the list may be either empty, i.e. `[]`, or else a non-empty list whose first element is some action (`A`, `L`, ...), and whose remainder is the list of unprocessed actions (`as`). The second case is shown as the pattern `(A:as)` etc., which deconstructs the list with `A` as the first element,

*Of course, it would not be a major exercise to change this definition to actually drill a board.

case	`[]`	`(A:)`	`(L:)`	`(R:)`	`(U:)`	`(D:)`
measured	2512	1563	1533	1728	1932	2137
manual	2970	1891	1855	2109	2363	2617
/measured	1.182	1.210	1.210	1.220	1.223	1.225
automatic	3075	2033	2006	2269	2532	2795
/manual	1.035	1.075	1.081	1.076	1.072	1.068
/measured	1.224	1.301	1.309	1.313	1.311	1.308

Table 1: WCET Measurements and Analysis for each Case

binding the variable `as` to the rest of the list. The action list is processed element by element. There are three basic cases:

`[]` – the action list has been completely processed – we return the accumulated list of drilling positions, `dps`, paired with a control flag (calculated using the `pos_ok` function) that indicates whether or not the robot has returned to its initial position;

`A:as` – the current action is a drilling operation – having done this, we move on to the next action by calling `step` recursively on `as`, using the `((xpos,ypos):dps))` expression to place the current position, `(xpos,ypos)`, at the front of the list of drilling positions, `dps`;

`L:as, R:as, U:as, D:as` – the current action is a move – either `xpos` or `ypos` is changed as required, and we move on to consider the rest of the actions (`as`) recursively.

In the example above, there will be five recursive calls to `step`[†], starting with the `R` case, and ending with the `[]` case. The result will be the list of drilling positions, `((1,1):[])`, paired with an indicator that the final position is OK.

4.2 Amortised WCET Analysis

As in the queue example, amortisation allows us to avoid calculating the worst case cost of a list of actions as the length of the list multiplied by the worst case cost of any individual action. Our amortised analysis gives a bound for the worst-case execution time T_{WCET} in clock cycles depending on the number of occurences $\#X$ of each action constructor X in the input, which is $T_{\text{WCET}} \leq 3075 + 2033 * \#\texttt{A} + 2006 * \#\texttt{L} + 2269 * \#\texttt{R} + 2532 * \#\texttt{U} + 2795 * \#\texttt{D}$. This information is even more precise than expressing the WCET in terms of the list length, but if we wanted to reduce the information to this form, we could assume that $\#\texttt{U} = \#\texttt{D}$ when the robot returns to its initial position, i.e., $T_{\text{WCET}} \leq 3075 + \lfloor 2663.5 * n \rfloor$ for a list of length n.

Table 1 compares the bound for each constructor with (a) the corresponding bound obtained from a manual analysis of a trace of the function `step`, summing the costs of all HAM instructions; and (b) the measured WCET. The absolute values given in the table refer to times in terms of clock cycles on the Renesas M32C/85U processor. Even for the prototype implementation of our analysis, we achieve manual analysis results within 23% of the measured WCET and automatic analysis results within 8% of our manually obtained results.

5 Conclusions

Amortised WCET analysis offers a way to construct costs that abstract over individual operations, and also to specialise WCET costs for particular input cases. Using our amortised WCET analysis approach for source-code analysis combined with information from the AbsInt **aiT** tool for machine-code analysis, we have been able to produce guaranteed WCET bounds that are reasonably close to measured WCETs (and indeed average-case times) for a real embedded systems platform. We are now working on tightening the bounds on WCET by improving the output of the automatic analysis and the quality of the code that is generated for HAM instructions. We are also applying the analysis to larger examples, e.g. ones taken from image processing or autonomous vehicle control [12]. Finally, we are

[†]The compiler will actually optimise these to be tail-recursive; effectively introducing a `goto` analogously to the implementation of an iterative loop in C, so there is no significant cost associated with this recursion.

working on exploiting high-level control flow information to guide the **aiT** analysis of the low-level compiled code, and so to obtain improved bounds.

We have explained our approach in terms of Hume, and have developed our initial analyses in the same context. Hume is a research notation that allows us to focus on core cost issues without distraction by many of the features that are found in production languages. However, now that the fundamentals have been properly worked out in this setting, the general static analysis techniques used here could, *in principle*, be applied to any other high-level language, including C. Moreover, the principle of amortisation could be applied to many forms of WCET analysis, not just the guaranteed WCET analysis we have described here.

Acknowledgements

This work is generously supported by EPSRC grant EP/C001346/1, by EU Framework VI IST-510255 (EmBounded) under the FET-Open programme, by an SOED support fellowship from the Royal Society of Edinburgh, and by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

References

[1] G. Bernat, A. Colin, and S.M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. 23rd IEEE RTSS 2002*, Austin, TX. (USA), December 2002.

[2] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Proc. IFL 2006*, LNCS 4449. Springer-Verlag, 2007. To appear.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] Renesas Corp. http://www.renesas.com. 2007.

[5] AbsInt GmbH. http://www.absint.com. 2007.

[6] K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Proc. First Central European Summer School, CEFP 2005*, LNCS 4164, pages 100–134. Springer-Verlag, 2006.

[7] K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hofmann, S. Jost, H.-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Towards formally verifiable resource bounds for real-time embedded systems. *ACM SIGBED Review— Special issues, ITCES06*, 3(4):27–36, October 2006.

[8] K. Hammond, G. Michaelson, and P.B. Vasconcelos. Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach. *ACM TOSEM*, 2007, under revision.

[9] K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pages 37–56. Springer-Verlag, 2003.

[10] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *ESOP 2006*, LNCS 3924, pages 22–37. Springer-Verlag, 2006.

[11] S. Jost, H.-W. Loidl, K. Hammond, M. Hofmann, and A. Bonenfant. Generic amortised resource analysis for higher-order functional programs. In Preparation, 2007.

[12] G. Michaelson, A. Wallace, K. Hammond, I. Wallace, A. Bonenfant, and Z. Chen. Towards resource certified image processing software. In *SEAS DTC Annual Technical Conference, July 2006, Edinburgh, Conference Proceedings*, page A15. UK MoD, 2006.

[13] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[14] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. 5th Intl Workshop on WCET Analysis*, pages 21–24, 2005.

[15] IAR Systems. http://www.iar.com. 2007.

[16] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. Accepted for ACM TECS, 2007.

Clustering Worst-Case Execution Times for Software Components

Johan Fredriksson*, Thomas Nolte, Andreas Ermedahl, Mikael Nolin
Dept. of Computer Science and Electronics
Mälardalen University, Västerås, Sweden

Abstract

For component-based systems, classical techniques for Worst-Case Execution Time (WCET) estimation produce unacceptable overestimations of a components WCET. This is because software components more general behavior, required in order to facilitate reuse. Existing tools and methods in the context of Component-Based Software Engineering (CBSE) do not yet adequately consider reusable analyses.

We present a method that allows different WCETs to be associated with subsets of a components behavior by clustering WCETs with respect to behavior. The method is intended to be used for enabling reusable WCET analysis for reusable software components. We illustrate our technique and demonstrate its potential in achieving tight WCET-estimates for components with rich behavior.

1 Introduction

In this paper we present a method that allows reuse of components with rich behavior in contexts where not all functionality of the components is needed. Typically, software components with rich behaviour have a worst-case execution time that may be drastically overestimated when the component is applied in a specific context. For these contexts it is imperative to be able to analytically reduce the estimated resource usage in order to achieve tight predictions of high quality. Thus increasing accuracy of predictions.

The work presented in this paper is intended to facilitate reusable WCET analysis for software components, e.g., in the framework presented in [1, 2].

Components are often reused over product boundaries, i.e., they are part of product lines and it is desirable to use the same component without re-analysis or recompilation. However, different products offer different contexts or usage of components; thus a component used in, e.g., a truck, may use different parts of the component compared to the same component used in a caterpillar. Using a context insensitive WCET analysis may be very inaccurate compared to the actual WCET^{truck} or $\text{WCET}^{caterpillar}$ (WCETs of the truck and caterpillar respectively), leading to a poor utilization of the system resources because of large differences between predicted behavior and actual behavior.

Resource constraints and predictability requirements are especially common in embedded-systems sectors, such as automotive, robotics and other types of computer controlled equipment. Because of the intrinsically non-linear behavior of software, it is often hard to make accurate predictions of the WCET of a piece of software. The problem is worsened in component-based development where components are kept independent of context to facilitate reuse. It is desirable to have an accurate analysis, allowing for the implementation of a system with less resources. This can be achieved by considering the context in which the software is used.

The contribution of this paper is a method for increasing the accuracy of a component's WCET by clustering execution-times with respect to usage. We use binary search heuristics to efficiently create clusters of similar execution-times. We describe and formalize the method, and exemplify with an illustrative example. Finally we use a simple academic case study and create clusters of two components.

The outline of the rest of this paper is as follows; in Section 2 we discuss related works. Usage scenarios are discussed in Section 3. In Section 4 component WCET analysis and the WCET clustering method are presented. In Section 5 we evaluate the method. In Section 6 we discuss the applicability of the method, and finally, Section 7 concludes the paper and future work is discussed.

2 Related work

Static WCET analysis is the only safe method for estimating WCETs for hard real-time systems [3]. However, traditional static WCET analysis does not consider usage. Software components designed for reuse are often more general compared to application specific code, leading to that parts of the component are only used in specific usages; in turn leading to greater variance of execution times. For component-based systems, where reuse is in focus, it is

*contact author: johan.fredriksson@mdh.se

desirable to not being forced to reanalyze components for each usage, at least within the same platform.

One approach to solve similar problems is parametric WCET. This has been proposed by many researchers within the WCET community but there is still very few parametric WCET methods developed. In [4] Björn Lisper outlines a technique for fully automatic parametric WCET analysis, which is based on known mathematical methods. In a MSc thesis [5, 6] a method inpired by Lispers work has been developed and tested with the aiT tool [7]. However, the focus of this work is not reusable WCET analysis, and reanalysis is required for different usages. A program representation for parametric WCET analysis has been suggested by Colin and Bernat [8]. Vivancos et. al. [9] propose an iterative method for computing WCET for loops parameterized in the number of loop iterations.

The main differences between the method proposed in this paper compared to parametric WCET is that we try to find execution-times for given input domains with the aim to create clusters of inputs that result in similar execution-times. Hence, a cluster is repersented by conditions on the inputs and a execution-time. A usage profile (limitations on inputs) is subsequently applied to the clusters to assess which clusters, and thereby which execution-times, that must be considered to be the WCET.

In [10] each basic block of a program is analyzed with respect to execution times and probability distributions of the execution times are derived. This method is, in comparision to our method, based on measurements. In [11] a framework has been developed that considers the usage of a system; however, neither software components nor reuse is considered. In [12] the source code is divided in modes depending on input, and only modes that are used in a given context is analyzed. In [13] a framework for probabilistic WCET with static analysis is presented. The probabilities are related to the probability of possible values of external and internal variables. All mentioned methods have the drawback of requiring reanalysis for every new usage.

Recent case-studies show that it is important to consider mode- and context-dependent WCET estimates when analyzing real sized industrial software systems [14, 15].

There are several WCET tools that support assertions and conditions to make the WCET tighter, e.g., aiT [7], RapiTime [16], Bound-t [17] and SWEET [18].

3 Usage scenario

In the "real" physical world, distinct modes exist and are often engineered into systems, for example, as *modes of operation*. We hypothesize that modes are significant discriminators of WCET and can be utilized for more accurate WCET modeling.

In [19] usage scenarios are probability distributions for so-called modes. Probabilities are estimated using large number of long program runs. To guarantee statistical properties (for example relative independence of input order), the program runs are divided into short runs, for example cycles in periodic real-time systems, transaction in transaction processing systems, and if necessary sampled. Modes are then defined as sets of similar runs based on input classes or other context parameters.

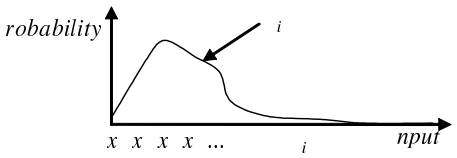


Figure 1. Input variable I.

Thus we define a *usage scenario* as $U = \langle X_0, ..., X_{n-1} \rangle$, where the $X_i (0 \leq i < n)$ are input variables, each with bounds on values, a given type, and a probability distribution $P_i : X_i \rightarrow [0, 1]$ for the occurrence of these values in the input. We assume that these variables (and hence their distributions) are chosen to be statistically independent and either have small domains naturally or model discredited partitions of real input variables. (See Figure 1 for an illustration of these concepts). The input domain M is then defined as $M := X_0 \times \cdots \times X_{n-1}$. The probability distributions $P_i (0 \leq i < n)$ extend uniquely to a probability distribution $P : M \rightarrow [0, 1]$ on the input domain, defined by $P(x_0, \ldots, x_{n-1}) = P_0(x_0) \times \cdots \times P_{n-1}(x_{n-1})$.

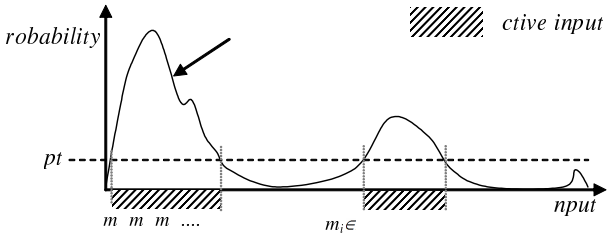


Figure 2. Usage scenario.

Furthermore we assume that $0 \leq pt < 1$ is a given probability threshold for ignoring low probability inputs (and consequently later their times). This will permit predictions of the form "with 0.99 probability WCET$< 500ms$." Inputs over the threshold are called *active* and the ratio of *active inputs* over *all inputs* is called the usage-scenario *utilization*. See also Figure 2 for an illustration of the concept.

4 Component WCET analysis

Components are reused in different products and different contexts. A different usage profile can substantially change the behavior of a component. To predict the execution time of a complex component with high accuracy, components must today be reanalyzed for every new usage profile – a very costly activity. Furthermore, it is not certain that the source code is available for components as they may be delivered by sub contractors. In this case analyses become even more costly [20].

Our method overcomes the problem by analyzing the execution times and their probability as a function of the input of the component. We assume that execution time varies with different inputs and their associated modes.

We define an input domain $\mathbf{I}$ for a set of input variables $\{X_0, X_1, \ldots, X_{n-1}\}$ as $\mathbf{I} = X_0 \times X_1 \times \cdots \times X_{n-1}$. Each element q in $\mathbf{I}$ is associated with an execution time $ET(q) \in \mathbf{W}$, where all execution times of the component are represented in the set $\mathbf{W}$. The longest execution time $\max(\mathbf{W}) = \text{WCET}^{abs}$ is the absolute WCET. A traditional static WCET tool will only find an estimate $\text{WCET}^{est} \geq \text{WCET}^{abs}$; however, we want to find the WCET for a specific usage. Because $\mathbf{I}$ often is very large, we can not perform WCET analysis for every element in $\mathbf{I}$ (every possible usage), instead we perform static WCET analysis with annotations on the input parameters, and perform a number of systematic runs with different bounds on the input parameters. When WCET analysis is performed with restrictions on the input parameters, not *all* input elements are considered, but rather a set of clusters $\{\mathbf{D}_l | \mathbf{D}_l \subseteq \mathbf{I}\}$, such that $\mathbf{D}_0 \oplus \mathbf{D}_1 \oplus \cdots \oplus \mathbf{D}_{n-1} = \mathbf{I}$, where $\mathbf{A} = \mathbf{B} \oplus \mathbf{C}$ means $\mathbf{B} \cap \mathbf{C} = \emptyset \wedge \mathbf{A} = \mathbf{B} \cup \mathbf{C}$. Thus, a cluster is a subset of all possible inputs, and a WCET tool can produce a WCET considering only that subset of inputs. Each cluster $\mathbf{D}_l$ is analyzed and associated with two execution times $et_l^{max} = \max(ET(d))_{d \in \mathbf{D}_l}$ and $et_l^{min} = \min(ET(d))_{d \in \mathbf{D}_l}$. The time et_l^{max} is the result of running the WCET tool with the inputs represented in $\mathbf{D}_l$ with respect to WCET. The time et_l^{min} is the result of running the WCET tool with the inputs represented in $\mathbf{D}_l$ with respect to best-case execution time (BCET).

As with all static WCET analyses all execution time estimates are safe over-estimations.

4.1 Clustering WCETs

To handle the size of the input domain $\mathbf{I}$ clusters need to be expressed with bounds or other operators, where each bound is associated with a WCET. It is often unfeasible to make a list of all inputs that are associated with one cluster; furthermore, WCET-tools often uses bounds to restrict the inputs. With the mathematical operators $\{\leq, >\}$ ranges of inputs can be expressed. The clusters $\mathbf{D}_l$ should be chosen in such a way that similar execution times are grouped and can be expressed as restrictions on the inputs. A challenge is to find the right clusters $\mathbf{D}_l$ such that accuracy of execution times become high.

4.2 Finding clusters

When the input domain $\mathbf{I}$ is too large to perform WCET analysis for every single input combination it is necessary to divide $\mathbf{I}$ into clusters of input combinations and analyze each cluster with respect to execution time. As the relation between inputs and WCET is not known a priori, the input space must be searched to find clusters such that all input combinations within the cluster produces similar execution times. In order to find such clusters it is necessary to have a way of evaluating clusters.

Theoretically, each single input combination has only one fixed execution-time. The difference between et_l^{max} and et_l^{min} of a cluster $\mathbf{D}_l$ shows the greatest difference between two execution times within the cluster. This in turn is an indicator of how similar the execution times are in the cluster. The sum of the difference between et_l^{max} and et_l^{min} of all clusters $\sum_l (et_l^{max} - et_l^{min})$ should be minimized to get the highest accuracy. In the extreme, each cluster contains one element; a good solution is a trade-off between acceptable difference and max number of clusters. If the difference between et_l^{max} and et_l^{min} of the cluster is larger than the required accuracy the cluster is not evaluated as a good cluster. Thus, the allowed difference between et_l^{max} and et_l^{min} of the cluster depends on the required accuracy of the cluster.

It is desired to create as few clusters as possible and yet acquire as high accuracy as possible. Clusters are effectively annotations (input restrictions) to a WCET-tool. Hence, we need methods to find annotations for WCET-tools.

To find accurate clusters with the least effort we propose a binary tree search approach, recursively dividing the input space into two clusters until the required accuracy has been found for all branches. Finding the clusters is a blind search problem. The only data initially known is the longest and shortest execution time for the entire search space (the WCET and BCET). This lack of knowledge depends on the nature of most WCET-tools, they provide a WCET and a BCET given a program and annotations; we want a large number of execution times considering different input combinations. The more the input space is divided the more data become available. There are several possible approaches to solve blind search problems, where binary search, simulated annealing and evolutionary search, are a few possible candidates.

Consider a simple example (Figure 3) with a function `foo` having two input variables x and y, where x can take the values $[0..9]$ and y can take the values $[0..4]$. All possi-

ble execution times given this simple example are summarized in Table 1. In this small example there are only 50 possible input combinations, and it is trivial to make an exhaustive search to find all combinations that give the same execution time. In a larger example, this is not possible. We have chosen such a simple example to simplify the visualization of the method.

```
// typeA:[0..9], typeB:[0..4]

typeA foo(typeA x, typeB y)
{
    if(x > 2 && y < 3)
        x=bar1(x,y);          // 30ms

    if(x > 2 && y > 2)
        return x-y;           // 1ms

    if(x < 5 && y > 0)
        x=bar2(y,x);          // 40ms

    if(x == y)
        return bar3(0,0);     // 20ms


    return bar4(x+y,x-y);     // 100ms
}
```

Figure 3. Example code.

#i	x	y	cond.	et_l^{max}	et_l^{min}
4	$[3,4]$	$[1,2]$		170	170
2	$[3,4]$	$[0]$		130	130
12	$[0,2]$	$[0,4]$	$x \neq y$	140	140
15	$[5,9]$	$[0,2]$		130	130
2	$[1,2]$	$[1,2]$	$x = y$	60	60
1	$[0]$	$[0]$		20	20
14	$[3,9]$	$[3,4]$		1	1

Table 1. Clustered WCETs with respect to the example code shown in Figure 3. #i is the number of input combinations. x and y are the limitations on the inputs. Cond is a logical condition on the inputs and $\mathrm{et}_\mathrm{l}^{\mathrm{max}}$ and $\mathrm{et}_\mathrm{l}^{\mathrm{min}}$ are the longest and shortest execution times produced by the inputs.

One set of values produce the worst-case execution time WCET. In the example in Figure 3 the WCET is produced by inputs represented by the first row in Table 1. All other input combinations lead to lower execution times. Consider an example where the usage scenario defines $x = \{3..6\}$ and $y = \{3..4\}$, the WCET will never occur. A WCET topology of the example is shown in Figure 4. For the case of a 2-dimensional input domain, the WCET topology is visible in an execution time matrix as shown in Figure 5.

The initial knowledge of the matrix is only the highest and lowest values (Figure 6.a). Since the knowledge of

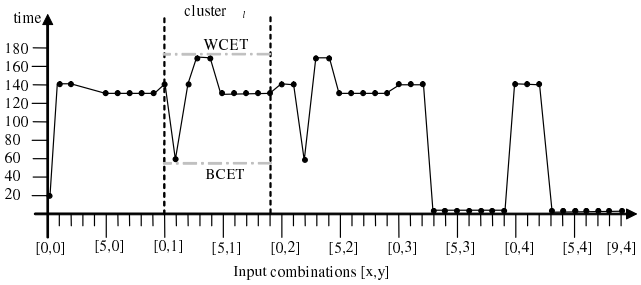


Figure 4. WCET topology with respect to the example code shown in Figure 3.

the execution times is limited we need a search method to localize areas with the similar execution times. One approach is to make a binary search for similar WCETs. In Figure 6 binary search is shown, dividing the search space into smaller and smaller clusters until the desired accuracy has been reached. The accuracy is defined as the distance between the highest and lowest values et_l^{max} and et_l^{min} for each cluster. In Figure 6, clusters that have reached their desired accuracy are marked with "*".

x \ y	0	1	2	3	4
0	20	140	140	140	140
1	140	60	140	140	140
2	140	140	60	140	140
3	130	170	170	1	1
4	130	170	170	1	1
5	130	130	130	1	1
6	130	130	130	1	1
7	130	130	130	1	1
8	130	130	130	1	1
9	130	130	130	1	1

Figure 5. Matrix of the inputs {x,y} with corresponding execution times with respect to the example code shown in Figure 3. The dotted line shows the cluster D_l as shown in Figure 4.

If the input space is divided into too few clusters accuracy will be lost; consider the extreme case of only using one cluster (all inputs), then the accuracy will be the same as standard WCET analysis. Due to large input spaces it is often infeasible to make an exhaustive search; therefore, even when the input domain is divided into a relatively large number of clusters it is still important how these are chosen

to maximize accuracy. Since the analysis is supposed to be reused, the effort of the analysis itself is of less concern.

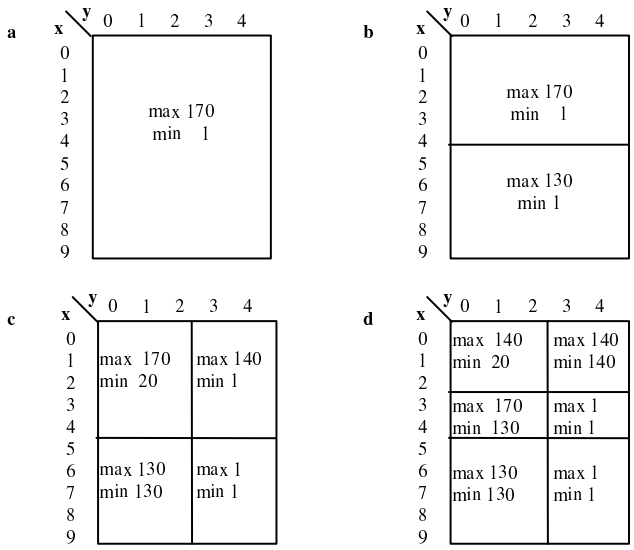


Figure 6. Binary search with respect to the input matrix shown in Figure 5. An '*' indicates that a cell does not need to be further divided. Max indicates the WCET reported by the tool given the annotations, and min the BCET dito.

5 Evaluation

We have performed a small evaluation with the SWEET WCET-tool [18]. SWEET has an annotation language to give restrictions on input parameters. Hence, it is very suitable for the approach presented in this paper. The annotations are described by the clusters.

Two components from an academic adaptive cruise controller (ACC) have been analyzed, "loggerOutput" and "SpeedControl". Both components have three input variables. We have a performed a guided binary search on both components. The guidance consisted of limitations on the input variables to 8 values for each input; these limitations were chosen based on the source code. The result of the guidance was an input domain of $8^3 = 512$ input combinations on each of the components. It required 12 clusters of the input domain of the "LoggerOutput" component to partition the execution times and produce 3 WCET expressions called *contracts*. The execution times were more scattered in the "SpeedLimit" component and it required 25 clusters to isolate all execution times into three contracts, The final contracts derived from the clusters for the "LoggerOutput" and "SpeedLimit" components are shown in Tables 2 and 3.

#	Expression	WCET
1	$i_2 \leq 0 \wedge i_3 \leq 0$	239
2	$(i_2 > 0 \wedge i_3 \leq 0) \vee (i_2 \leq 0 \wedge i_3 > 0)$	433
3	*other*	627

Table 2. LoggerOutput component "contract" from 12 clusters.

#	Expression	WCET
1	$i_1 \leq 0$	105
2	$i_2 > 0 \wedge ((i_1 = 0 \wedge i_3 < 0) \vee$ $(i_1 > 0 \wedge i_3 \geq 0))$	384
3	*other*	263

Table 3. SpeedLimit component "contract" from 25 clusters.

The derived contracts are used with a usage scenario on the input parameters. Depending on the usage the contracts will give the WCET corresponding to the usage.

We see that for many usages we get substantially lower WCETs for both components. Using a traditional usage independent analysis would produce much to pessemistic WCET for many usage scenarios.

6 Applicability

The method described in this paper is a general clustering method that is well suited for creating contract based WCETs for components. Each cluster can also be augmented with more information, e.g., scheduling parameters and information on energy consumption. In this way clusters can be created with respect to several parameters and trade-offs between them can be made.

Furthermore, the proposed method is useful for both hard and soft real-time systems. In this paper we have only described the application for hard real-time systems.

The methods as described in this paper indirectly perform an exhaustive WCET analysis because all input combinations are represented. This will result in safe overestimations and the "real" WCET is guaranteed to be included in the analysis.

For soft real-time systems, a number of input combinations (not clusters) can be analyzed with respect to execution-time and clusters can be created through, e.g., the least square method.

The focus of the method is still to create tight and accurate *reusable* WCET estimations through expressing the WCET as usage parameterized contracts.

6.1 Hardware effects

It should be noted that the contracts specified for the clusters only consider input data limits. The timing of the code in the cluster will also be dependant on the hardware upon which the code is executed and where in memory the code is located. Assuming that a simple 4-, 8- or 16-bit CPU is used, which is common in a large segment of the embedded domain, and that the code is forced to reside in and access memory areas with the same timing properties as assumed in the WCET analysis, the WCET estimates derived should also be valid in the new context. However, if a more advanced CPU is used, maybe with a cache or some other performance enhancing features, and/or if the compiler and linker change the code structure, and/or if some other hardware timing properties are changed, the derived component WCET estimates should be used with caution. Thus, in the latter case the contract for a component might also need to include information upon the hardware, compiler and linker configuration. This is something not yet considered in our work.

7 Conclusions and future work

Component-Based Software Engineering (CBSE) is a promising development method to reduce time-to-market, reduce development costs, and to increase software quality. One main characteristic of CBSE that enable these benefits is its facilitation of software component reuse, i.e., the same software component can be used in different contexts. Unfortunately for resource constrained systems, reusable components with rich behavior increase resource consumption by decreasing the tightness of analyses.

In this paper we have presented a method for clustering Worst-Case Execution-Times (WCETs) with respect to behavior for reusable software components. The purpose of the method is to associate different WCETs with subsets of the component behavior to achieve tight WCET estimates. The presented method is intended to be used for facilitating reusable WCET analysis for reusable software components as presented in, e.g., [1, 2]. We have illustrated the method and demonstrated its potential in a small case study.

Future work includes case studies on large components to evaluate the feasibility of the approach. Also case studies on industrial code is planned to evaluate the industrial appropriateness of the proposed method. We also plan to investigate augmentation of clusters with additional parameters, e.g., scheduling parameters.

References

[1] Fredriksson, J., Nolte, T., Nolin, M., Schmidt, H.: Contract-based reusable worst-case execution time estimate. In: Proc. of the 13^{th} International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07), Daegu, Korea (2007)

[2] Fredriksson, J., Land, R.: Reusable component analysis for component-based embedded real-time systems. In: Proc. of the 29^{th} International Conference on Information Technology (ITI 2007), Cavtat near Dubrovnik, Croatia (2007)

[3] 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., Stenström, P.: The worst-case execution time problem — overview of methods and survey of tools. Accepted for publication in ACM Transactions on Programming Languages and Systems (2007)

[4] Lisper, B.: Fully automatic, parametric worst-case execution time analysis. Technical report, Mälardalen Real-Time Research Centre (2003)

[5] Altmeyer, S.: Parametric wcet analysis, parametric framework and parametric path analysis. Master's thesis, Saarland University, Department of Computer Science (2006)

[6] Humbert, C.: Parametric wcet analysis, parameter analysis and parametric loop analysis. Master's thesis, Saarland University, Department of Computer Science (2006)

[7] aiT: (ait execution time analyzer) Absint: http://www.absint.com/ait/.

[8] Colin, A., Bernat, G.: Scope-tree: A program representation for symbolic worst-case execution time analysis. In: ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems, Washington, DC, USA, IEEE Computer Society (2002) 50

[9] Vivancos, E., Healy, C., Mueller, F., Whalley, D.: Parametric timing analysis. In: LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, New York, NY, USA, ACM Press (2001) 88–93

[10] Bernat, G., Colin, A., Petters, S.: pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems. In: WCET. (2003) 21–38

[11] Lee, J.I., Park, S.H., Bang, H.J., Kim, T.H., Cha, S.D.: A hybrid framework of worst-case execution time analysis for real-time embedded system software. In: Aerospace Conference, IEEE (2005) 1–10

[12] Ji, M.L., Wang, J., Li, S., Qi, Z.C.: Automated wcet analysis based on program modes. In: Proc. of AST'06, Shanghai, China, ACM (2006)

[13] David, L., Puaut, I.: Static determination of probabilistic execution times. In: Proc. of the 16^{th} Euromicro Conference on Real-Time Systems (ECRTS'04). (2004) 223–230

[14] Sehlberg, D., Ermedahl, A., Gustafsson, J., Lisper, B., Wiegratz, S.: Static wcet analysis of real-time task-oriented code in vehicle control systems. In: Proc. of the 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06), Paphos, Cyprus (2006)

[15] Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying static wcet analysis to automotive communication software. In: Proc. of the 17^{th} Euromicro Conference of Real-Time Systems, (ECRTS05), Mallorca, Spain (2005)

[16] RapiTime: (Rapitime execution time analyzer) Rapita Systems: http://www.rapitasystems.com/.

[17] Bound-t: (Bound-t execution time analyzer) Tidorum Ltd: http://www.tidorum.fi/bound-t/.

[18] SWEET: (Swedish execution time tool) SWEET: http://www.mrtc.mdh.se/projects/wcet/.

[19] John D. Musa, A.I., Okumoto, K.: Software Reliability - Measurement, prediction, application. McGraw-Hill, New York (1987)

[20] Korel, B.: Black-box understanding of cots components. In: Proc. of the 7^{th} International Workshop on Program Comprehension (IWPC '99), Washington, DC, USA, IEEE Computer Society (1999) 92

Tighter WCET Estimates by Procedure Cloning*

Paul Lokuciejewski, Heiko Falk, Martin Schwarzer, Peter Marwedel
Embedded Systems Group, Dept. of Computer Science 12
University of Dortmund
D-44221 Dortmund, Germany
{Paul.Lokuciejewski | Heiko.Falk | Martin.Schwarzer | Peter.Marwedel}@udo.edu

ABSTRACT

Embedded software spends most of its execution time in loops. To allow a precise static WCET analysis, each loop iteration should, in theory, be represented by an individual calling context. However, due to the enormous analysis times of real-world applications, this approach is not feasible and requires a reduction of the analysis complexity by limiting the number of considered contexts. This restricted timing analysis results in imprecise WCET estimates. In particular, data-dependent loops with iteration counts depending on function parameters cannot be precisely analyzed. In order to reduce the number of contexts that must be implicitly considered, causing an increase in analysis time, we apply the standard compiler optimization *procedure cloning* which improves the program's predictability by making loops explicit and thus allowing a precise annotation of loop bounds. The result is a tight WCET estimation within a reduced analysis time. Our results indicate that reductions of the WCET between 12% and 95% were achieved for real-world benchmarks. In contrast, the reduction of the simulated program execution time remained marginal with only 3%. As will be also shown, this optimization only produces a small overhead for the WCET analysis.

1. INTRODUCTION

Real-time systems acting in a safety-critical environment must meet timing constraints imposed by the system specifications which are based on the knowledge of the worst-case execution time (WCET). This key parameter can be calculated in several ways.

One technique is the static WCET analysis determining upper timing bounds of a program. Besides the hardware timing characteristics specifying the execution time of single instructions, the analysis relies on *flow facts*. They can be divided into two classes, namely the mandatory ones and those that improve the timing analysis. The mandatory flow facts serve as restrictions to overcome the halting problem by defining the iteration counts of loops and the recursion depth and thus to ensure that the program will terminate [11]. The second class contains information used to describe the program structure more accurately, in particular flow facts identifying *infeasible paths* that are potentially executable according to the control flow graph but are not feasible due to the program semantics and the given input data. This information is not mandatory to obtain a safe WCET estimate but might improve its tightness [7].

Embedded software spends a large amount of its execution time in loops. Thus, the WCET analysis of loops is crucial and strongly relies on precise flow facts describing the number of loop iterations. To improve the analyzability of loops, each single iteration is represented by a context to explicitly model a particular state of the loop execution. When the number of contexts during the WCET analysis is not restricted and each loop iteration is assigned a separate context, the resulting WCET estimates are tight.

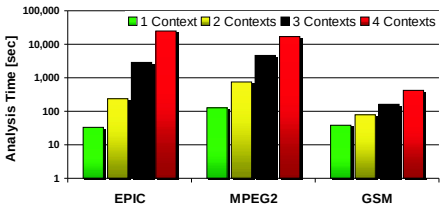


Figure 1: Context-Sensitive WCET Analysis Times

Real-world applications, however, are too complex to allow an efficient WCET analysis where each emerging context is taken into account. Figure 1 addresses this issue. For three complex benchmarks from the commonly used *MiBench* suite [8], a WCET analysis was performed using aiT [1], a static WCET analyzer developed by AbsInt. The number of distinguished contexts was varied from one to four for all benchmarks. As can be seen, the analysis time increases rapidly. For example, the analysis time for the MPEG2 benchmark took 125.79 seconds for one and 16723.22 seconds for four distinguished contexts. Despite these long analysis times, the WCET estimates are still highly overestimated since the small number of considered contexts was not sufficient for a precise timing analysis.

To cope with the exploding analysis time, the number of contexts must be restricted. A restriction to n contexts means that the first $n-1$ loop iterations are considered separately while the remaining loop iterations are summarized into a single context. This has a negative impact on the WCET estimation since loops cannot be analyzed precisely and worst-case assumptions must be done for the summarized context to guarantee safeness.

In particular, these assumptions might lead to large WCET overestimations for typical embedded systems applications which contain functions with loops whose number of iterations depend on function parameters. When the function is called multiple times with different arguments controlling the number of loop iterations, each loop execution requires individual loop bounds. State-of-the-art timing analyzers handle this issue by performing a loop analysis to determine explicit numbers of loop iterations for each loop execution. The approach enables a tighter WCET analysis since each loop contributes with its realistic runtime to the global WCET.

However, the detection of loop bounds succeeds only for simply structured loops. In order to provide the essential in-

*This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/)

formation about the unrecognized loop iteration counts, the user must provide these annotations manually. This is done by defining an interval for each loop, modeling lower and upper bounds [6]. Any iteration counts lying in between the boundaries cannot be explicitly specified. Hence, exact loop bounds cannot be derived for a particular loop execution from these user annotations. This ignorance is an inherent source of imprecision since WCET analyzer must assume the worst case and consider each loop execution exclusively with the specified upper loop bounds regardless of their real behavior. To solve this problem, we propose procedure cloning to make loops more explicit.

Procedure Cloning (also known as *Function Specialization*) is a standard compiler transformation [10] optimizing functions called with different constant arguments. These functions are cloned, the constant parameters are removed from the parameter list and their occurrence is directly replaced by the constant values. The well-known benefits concerning the average-case execution time (ACET) are a simplified code that offers opportunities for further optimizations like constant folding as well as a reduced calling overhead.

Compared to the marginal improvements of the ACET, procedure cloning can be highly efficient for the WCET analysis since it makes different function call contexts explicit and thus allows a separate loop annotation for each context. This paper is the first to exploit this compiler optimization to improve a programs predictability. Loops that are data-dependent of constant function arguments now become better analyzable since their original variable loop bounds are replaced by constants allowing an explicit annotation within each cloned function. Hence, the optimized code offers the opportunity to calculate tighter upper timing bounds in an acceptable amount of time while the number of contexts remains restricted.

In addition, the tightness of the WCET estimates can be further improved by the elimination of infeasible paths. In the non-optmized code, loops might contain paths that are only traversed for a small number of loop executions. A safe WCET analysis must assume the worst case for the summarized context and also considers these paths even though they are never taken in reality for this particular calling context.

The rest of this paper is organized as follows: Section 2 describes related work. In Section 3, the concepts behind calling contexts and the resulting loss of analysis precision due to their restriction are discussed. Procedure cloning and its benefits for WCET analysis are presented in Section 4. Section 5 describes the experimental environment, followed by benchmarking results in Section 6. Section 7 summarizes the contributions of this paper and gives directions for future work.

2. RELATED WORK

Procedure cloning has been introduced by Cooper [3] and is nowadays part of many optimizing compilers [10]. Up to now, this approach was considered in the context of ACET and the main objective was the increase of the average-case performance while keeping the resulting code size increase small. This paper, in contrast, studies the benefits of procedure cloning on the WCET analysis and exploits them for an efficient and precise timing estimation.

In the previous decades, research on compiler optimizations mainly focused on the reduction of the average-case execution time. With the growing importance of embedded systems, other criteria like energy dissipation or code size become significant and various approaches have been presented describing how to use compiler optimizations to reduce them. However, there was little research on the minimization of the WCET by a compiler. The approach described in [5] is one of the few examples where a compiler optimization developed for ACET reduction was employed to study its influence on the WCET.

In [12], a code-positioning optimization driven by worst-case path information was presented. The algorithm rearranges the memory layout of basic blocks to avoid branch penalties along the WC path. The modified code has an improved pipeline performance and results in a reduced WCET.

A compiler guided trade-off between WCET and code size for an ARM7 processor was studied by [9]. The authors observed that applications implemented with 16-bit THUMB instructions are smaller but also slower than the same code using the full 32-bit instruction set. They use a simplified timing analyzer to obtain WCET information employed in their code generator to produce code that exploits this trade-off and uses the two instruction sets for different program sections.

A design study for a homogeneous WCET-aware compiler was presented in [4]. The compiler generates input data for a timing analyzer, starts the WCET analysis and finally imports WCET-relevant information back into its data structures. With the compiler knowledge about the program and the gained timing data, WCET-aware compiler optimizations can be realized. The focus of that paper was the design of a WCET compiler framework and can be considered complementary to this present work.

3. RESTRICTED CONTEXTS

This section discusses the concepts of contexts with a special focus on the analysis of loops. It shows why restricted contexts can be fatal for a WCET analysis and the resulting loss of tightness will be illustrated by an example.

The use of contexts is a common approach for static program analyses. It enhances the analysis of functions by representing each function call as an individual context. Thus, all program details like the passed arguments can be explicitly specified. To depict a function call invoked within another function, its calling history must be considered. This is achieved by specifying this particular context with a call string describing the sequence of functions called previously. Loops resemble recursive functions calls since they invoke themselves. To exploit the concepts of contexts for loops, and thus to enhance the precision of their timing and stack analysis, loops are transformed into dedicated functions calling themselves. After this loop transformation, each loop iteration is assigned an unique context.

As indicated in Section 1, real-world applications possesse a structure too complex to analyze each emerging context. In particular, nested loops can cause a rapid increase of contexts (*state explosions*). For example, the loop nest of three nested loops with ten iterations each is represented by 1000 contexts. Such large numbers of contexts make a static analysis infeasible since both a vast amount of memory resources and computation time are required. The only way out is to reduce the complexity of the static analysis by restricting the number of distinguished contexts.

In general, restricting the number of contexts simplifies the calling history of function calls by restricting the length of the call string. A definition of maximally n distinguished contexts means that the first $n-1$ functions calls will be analyzed separately with all their details concerning the system properties at this particular point of the program. All

following calls, in contrast, are summarized into the last context. Thus, all precise calling information like the encountered value ranges during a particular call cannot be explicitly expressed. The result is an inappropriate input data for the value analysis required for a successful cache and pipeline analysis.

Besides the analysis of function calls, an enormous loss of tightness may also occur when loops are annotated manually and iterations are summarized into single contexts. The loss of tightness arises from the mandatory conservatism to guarantee the safeness of the timing results. If a loop is part of the WC path, its contribution to the global WCET is calculated as follows: the maximal iteration count specified by the user annotations is multiplied by the local WCET of the loop body and added to the WCET for the evaluation of the loop's termination condition.

This entails two potential problems. First, program functions containing loops whose loop bounds depend on the function parameters might be invoked with varying arguments. The result are function loops with different iteration counts and thus have individual WCETs. Due to the limited user annotations that simply define the lower and upper loop bounds, the real number of iterations cannot be derived. A conservative static timing analysis must expect the worst case and assumes for the summarized context that the loop is executed as often as defined by the user maximum. Second, the loop body might consist of different paths each with an individual execution time. The WCET analysis must again proceed pessimistically and assume that all loop iterations summarized into the last context traverse the WC path.

3.1 Exemplary loss of tightness

To emphasize the loss of precision, an exemplary scenario is presented in the following. Assume that the function f from the left-hand side of Figure 2 is invoked twice with the values 100 and 5 for the parameter n. Furthermore, it is assumed that the WCET of the path containing the if-block is twice as large as when this block is omitted and that the number of contexts is restricted to $k < 5$. With the safe user annotation defining the upper loop bound to be 100, the first call to the function f results in an overestimation since for the summarized context it is assumed that the remaining $100 - k$ loop iterations contribute with the local WC path which includes the if-block. The overestimation is even worse for the second call to f. Here, for the summarized context the static analysis must assume that the loop is not iterated k, but 100 times and that the remaining $100 - k$ loop iterations traverse the if-block being the WC path. The result is a safe but also unacceptably large. Due to its immense overestimation, the analysis result represents an upper timing bound that does not expresses the actual worst-case behavior of the given program.

To cope with the dilemma of shortening the analysis time by restricted contexts but still obtaining tight WCET estimations, the standard compiler optimization procedure cloning can be expoited. We will show in the next section how it improves the predictability by generating a program structure that allows a precise annotation of loop bounds.

4. PROCEDURE CLONING

Procedure cloning belongs to the class of inter-procedural compiler transformations where the optimizing compiler generates a specialized copy of the original procedure. Afterwards, the original function calls are replaced by calls to the newly created clones. The optimized code provides a more beneficial basis for aggressive inter-procedural dataflow analyses [3]. On the other hand, cloning often offers the opportunity for improved optimizations, particularly for constant propagation and folding, copy propagation and strength reduction. Also, entire paths might be eliminated when cloning yields conditions that can be evaluated by the compiler as always false and thus be never executed.

```
int f(float *x, int n, int p) {
 for (i=1; i<=n; i++) {
  x[i] = pow(x[i], p);
  if (i==10) {...} }
 return x[n]; }

int main(void) {
 return f(a, 5, 2); }
```

```
int f1(float *x) {
 for (i=1; i<=5; i++)
  x[i] = x[i]*x[i];
 return x[5]; }

int main(void) {
 return f1(a); }
```

Figure 2: Example for Procedure Cloning

Figure 2 demonstrates cloning of function f allowing improved optimizations across function call boundaries [2, 10]. Replacing the function parameters n and p by the constants 5 and 2, respectively, offers a significant amount of optimization potential for the cloned function *f1*. First, applying *strength reduction* allows the replacement of the expensive call to function *pow* by a multiplication. Second, the propagated constant value of n results in a simplified control flow graph. By exposing the value range of the loop induction variable i, it is known at compile time that the condition *(i==10)* will never become true. Thus, this infeasible path can be eliminated yielding a smaller number of instructions and a better pipeline behavior due to the reduced number of control hazards. Last but not least, the calling overhead is reduced. The decreased number of passed arguments minimizes the number of required instructions for both the caller and the callee.

Besides the improvements concerning the program runtime, the optimization has one drawback. Each specialized copy of the function body increases code size. In general, it is also not always permitted to remove the original function even if it is not called anymore in the optimized program. On general purpose systems there is no guarantee that this function might be called from another compilation unit not considered in the current optimization course and its removal would be illegal. In the domain of embedded systems, this restriction is usually not given and the removal of original functions can be performed more aggressively. The designer knows in advance what software will be running on the system and can thus definitely determine the functions never called from other modules than the one they are located in. These original functions can be removed after cloning without endangering the systems consistency.

Hence, this compiler optimization should be used with caution, and a trade-off between the resulting speed-up and the increased code size, especially in the domain of embedded system's with restricted memories, should be taken into account.

4.1 Selection of functions to be cloned

There are different strategies to define how extensively procedure cloning should be performed. Two factors are relevant for the optimization. First, the maximum size of the function permitted to be cloned must be specified. This parameter can, for example, be defined by the number of source code expressions found within the function. All functions that exceed this parameter are omitted and not considered for procedure cloning since they may possibly result in a too large code size increase.

The second factor guides the choice of functions to be cloned by setting constraints on the occurrence of the constant arguments. It defines how frequently a particular constant argument must occur within all calls of the function to be cloned. For example, the user might specify that constant argument values must be present in more than half of all function calls. When this frequency is not reached, it will not be considered for optimization and the function will not be cloned for this parameter. If the code size increase is crucial, the number of additionally generated functions must be kept minimal. The only candidates for cloning are functions that are called most of the time with the same constant argument. The extreme case is the choice of functions that are always invoked with the same constant value for a particular function parameter.

Procedure cloning is performed in three stages where each function is analyzed separately. In the first step, constant arguments and the number of their occurrences for each function parameter are collected. Hereafter, the collected arguments that do not meet the specified frequency are removed and omitted for procedure cloning. This is done by counting all function calls the considered argument is used in and comparing it to the number of parameter occurrences from the previous step. In the final stage, all constant arguments that were not removed are used for procedure cloning. The original function is cloned and assigned a unique function name. The specialized argument is removed from the parameter list and directly propagated into the code by replacing the parameter variables by the constant value. Finally, the original function calls within the source code are redirected to the cloned functions.

4.2 Improvements to the WCET

In addition to the previously mentioned pure code optimizations that yield a better code quality, procedure cloning yields a program structure that strongly improves the WCET analysis by making the code more predictable. It tackles the two major problems discussed in Section 3: the explicit specification of loop bounds and the elimination of infeasible paths that may otherwise contribute to the WCET for the sake of safeness. Both contributions of procedure cloning enhance the tightness of the estimates since they result in a more accurate description of the program behavior.

Typical embedded software is loop-dominated. As studies on MiBench benchmarks [8] pointed out, many loops are located in functions and their number of iterations is often specified by function parameters as shown on the left-hand side of Figure 2. These functions, in turn, are called multiple times with varying constant arguments resulting in strongly deviating execution times spent in the loops.

To statically analyze these loops, the timing analyzer must be provided manually with loop iteration counts. To preserve WCET safeness, the loops are annotated with the maximal number of iteration counts the loop is ever executed with, i. e. the annotations must represent the global maximum of iterations for this loop.

These loose loop annotations can be specified more precisely after procedure cloning. When a function is called multiple times with varying constant values that dictate the upper loop bound, this is exploited by the transformation (see right-hand side of Figure 2). The variables in the specialized functions are replaced by the individual constants and thus provide clones that are *dedicated* to individual loop executions. The new user loop annotations can focus on each specialized function explicitly and annotate their loops more realistically. During WCET analysis with restricted contexts, these loops contribute with their corrected maximal number of iterations. Thus, the transformation aims at making the code more predictable.

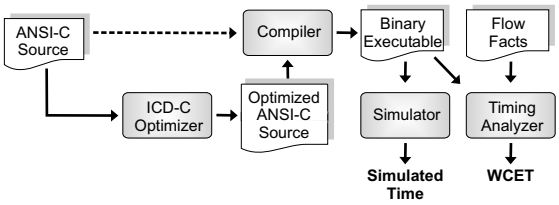


Figure 3: Workflow of Procedure Cloning

Yet another code simplification has a positive effect on the tightness of the WCET estimates. Loops often consist of multiple paths. Some paths may have the longest execution time (WC path in the loop) but are never executed due to unfulfilled conditions. A conservative timing analysis with restricted contexts must assume the worst-case scenario where each loop iteration represented by the summarized contexts goes through the WC path. After procedure cloning, these infeasible paths can be eliminated in the specialized functions.

This is illustrated by an example. As shown in function *f1* on the right-hand side of Figure 2, the path through the *if*-block is never traversed for parameter $n = 5$. Compiler data- and control-flow analyses are capable of detecting conditions that are evaluated as being always false and remove them from the control-flow graph. Thus, these infeasible paths are not taken into account during WCET analysis and don't unnecessarily contribute to the estimated upper timing bounds.

In the following sections, procedure cloning transformations described in this section are applied to real-world benchmarks and their improvements concerning the WCET estimates are presented.

5. EXPERIMENTAL ENVIRONMENT

This section describes the choice of benchmarks used to evaluate the influence of procedure cloning on the WCET. Furthermore, the benchmarking workflow is described.

The benchmarks come from the widely used MiBench suite representing different applications typically found in the embedded systems domain. The first benchmark is EPIC, an experimental lossy image compression utility. MPEG2 is a motion estimation for frame pictures, while GSM represents a speech compression.

All measurements were performed for two different 32bit processors. The first was an Infineon TriCore 1796 microcontroller. The other was an ARM7TDMI that supports two different instruction sets: a full set of 32bit instructions and the so called THUMB instruction set consisting of 16bit instructions. Both modes were exploited for the evaluation of the results.

The workflow is depicted in Figure 3. Two different binary executables are generated. One derived from the original code that is used as reference object (marked with the dotted line). The other binary is the resulting program after procedure cloning and the standard optimizations constant folding, constant propagation and dead-code elimination [5] to remove infeasible paths. The optimizations are automatically performed by a source-to-source optimizer (*ICD-C optimizer*). Our parameters for procedure cloning as described in Section 4.1 were a maximal function size of 2000 expressions and a frequency of 50% (constant argument to be cloned must occur in at least half of all function calls). The output of the optimizer, the optimized source code, is

provided as input for the compiler. Depending on the architecture under test, different commercial compilers were employed. For the TriCore processor the tricore-gcc was used. For the ARM7TDMI the compilers armcc and tcc for the ARM and THUMB mode, respectively, generated the binary executable.

In the last test phase, the binaries are passed to the cycle-true simulator to obtain the simulated processor cycles representing the program execution time for a typical input data set. In addition, the binary executables are passed to the timing analyzer aiT together with the configuration file containing the manually generated information about the loop bounds (flow facts). The result is the WCET for the evaluated TriCore and ARM instructions sets for both the original and the optimized code. Due to the complexity of the timing analysis as indicated in Figure 1, the number of contexts was restricted to 1 (no remarkable improvements were observed for two or three contexts).

6. RESULTS

WCET

Figure 4 depicts the relative WCET for the optimized code with respect to the WCET estimated before procedure cloning (corresponds to 100%). As can be seen, significant WCET reductions of up to 95% were achieved. In the following, the transformations performed by the ICD-C optimizer are briefly discussed for each benchmark and the resulting improvements are pointed out.

The WCET for the EPIC benchmark decreased by 94.61% for the TriCore processor. Similarly remarkable improvements were achieved for the ARM processor, namely 95.72% for the ARM mode and 95.65% for the THUMB mode. This is due to the code structure containing a large number of nested loops. The image coder benchmark contains a function that is highly appropriate for cloning. It is a filter containing 32 loops nested up to four times, and their number of iteration counts partially depends on the function parameters. Furthermore, the function is called six times with different constant values. After procedure cloning, each function call is specialized. The passed constants are propagated and in some cases explicitly define the upper loop bounds. The result is tighter *min* / *max* intervals for each loop execution e.g. $[1 \ldots 15]$ (in non-optimized code) becomes $[1 \ldots 1]$ after the transformation, meaning that the timing analyzer can assume one loop iteration in contrast to the pessimistic assumption of 15 iterations.

The benchmark MPEG2 contains two functions that were cloned. The first function implements the *Fullsearch* algorithm to detect the motion of macro-blocks. It is called with two different constant values defining the height of the image block. Within this function, another procedure is called computing the distance between these blocks. It is invoked with the same block height constants as passed to its caller. These values are used to control the number of iteration counts for multiple loops. The source-to-source optimizer performs cloning for each of these functions. The result is a transformed code that has a dedicated version of the Fullsearch implementation for each block size. The loop bounds in the nested function can again be defined more precisely. As for EPIC, the timing analysis of the loops becomes better analyzable and thus more predictable. This is confirmed by the benchmark results. For TriCore, the WCET after procedure cloning was reduced to 70.08% compared to the unoptimized code. Similar improvements were gained for the ARM processor: the worst-case execution time was reduced to 66.75% and 66.55% for the ARM and THUMB

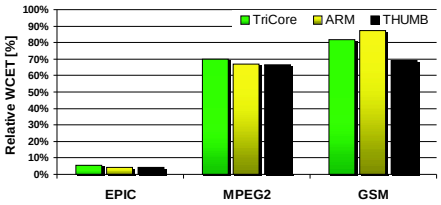


Figure 4: Relative WCETs after Procedure Cloning

modes, respectively. The reason for the strong reduction is the large number of function calls for the nested function. In total, it is called more than 1.4 million times. For the unoptimized code with imprecise loop bound specifications, each analyzed loop contributes to the overestimation.

GSM, the last evaluated benchmark, contains a function representing a filter for the short term residual signal. The function is called with strongly varying constants (13, 14 and 120) defining the number of iterations for its loop. Without procedure cloning, the designer must specify the loop bounds safely, and annotate the maximal number of loop iteration with 120. Obviously, for all calls with the constants 13 and 14, the timing results show an overestimation since the timing analyzer assumes 120 loop iterations. Procedure cloning solves this problem by cloning this function twice, one specialized version for the constant 13 and one for 120. Due to the small number of occurrences of the constant 14, this argument was not considered for specialization. The original version is kept and handles calls with the constant 14. Due to the improved analyzability, the loops can be exactly specified by the system designer. This has a positive effect on the estimated WCETs. Reductions from 12.73% (ARM mode) up to 31.03% (THUMB mode) with regard to the WCET of the non-optimized code were achieved.

The significant WCET reductions after procedure cloning comes basically from the possibility to tighten the *min* / *max* intervals specifying the loop bound iterations. Another reason for the success of this compiler transformation was the fact that the specialized functions were part of the WC path. Otherwise, their optimization would have had no effect on the WCET. As discussed in Section 4.2, the elimination of infeasible paths might also have positive effects on the timing results. However, for the benchmarks considered in this paper, they were marginal as comparisons between the original and optimized code indicated. Although some paths could be eliminated in the cloned function, they did not improve the WCET results since they had not lied on the WC path and were thus irrelevant for the timing analysis.

Simulated Program Execution Time

To examine the impact of procedure cloning on the ACET, the simulated execution times for typical program input data sets of the original and optimized code were compared. The improvements were negligible. For the EPIC benchmark, the simulated execution time even slightly increased between 0.02% and 3% for TriCore and THUMB mode, respectively. For MPEG2, the optimization gain was between 0.21% (for TriCore) and 2.6% (for the THUMB mode). No improvements were achieved for the GSM benchmark executed on the ARM7TDMI, and a minimal simulated time reduction of 0.01% was observed for the TriCore processor. The minimal degree of the execution time reduction came from the fact that the cloned functions did not provide additional opportunities to further improve the code by the performed source-to-source optimizations, i. e. the newly created functions did not allow to additionally simplify the code with the performed optimizations like constant folding.

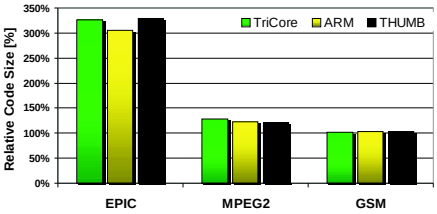


Figure 5: Relative Code Sizes after Cloning

Code Size

Finally, the code size is briefly examined. As mentioned in Section 4, the code size increase is a drawback. Each cloned function increases the code size in particular when the functions are large or multiple specialized copies of a function are created. In Figure 5, the relative code size is shown. 100% correspond to the code size of the benchmarks before procedure cloning. The code size of the optimized EPIC benchmark rose to more than 300% for all instruction sets since a large function with 32 nested loops was cloned six times. However, this increase is acceptable since the absolute code size of the optimized code remains small, namely 21 kilobytes. For MPEG2, the code size increased between 22.0% and 28.94% for TriCore and ARM7TDMI in the THUMB mode, respectively. Although procedure cloning created multiple specialized copies, they all were relatively small so that the code size increase was acceptable. For the GSM benchmark, the code size increase was negligible and achieved its highest value of 3.28% for the ARM mode. Again the reason is that just few and small functions were specialized.

Runtime of the WCET Analysis

To evaluate the impact on the complexity of the WCET analysis, all benchmarks were analyzed in their original and optimized version. The runtime of the timing analyzer aiT on an AMD Sempron 3000+ with 2 GB RAM on average increased by 14% for the analysis of the optimized code by procedure cloning compared to the analysis of the original code. The reason is a larger control-flow graph containing more functions than the original one. However, concerning the large WCET improvements, the increased analysis runtime is acceptable.

7. SUMMARY AND FUTURE WORK

This paper solves the problem of how to preserve the tightness of WCET results for programs with data-dependent loops that require a limited analysis with a restricted number of calling contexts. The analysis of these loops is an inherent source of unpredictability since their number of loop iterations can be rarely specified precisely. The resulting WCET is heavily overestimated. We propose procedure cloning as an approach to improve the analyzability and thus to make the code more predictable. The optimized code allows an explicit specification of loop bounds and furthermore advances the elimination of infeasible paths for tighter WCET results.

The effects of procedure cloning were evaluated with real-world benchmarks from the MiBench suite. The results emphasize the effectiveness of procedure cloning, a WCET reduction between 12% and 95% was achieved. In contrast, the simulated program execution time for the optimized code hardly changed after the optimization. The results also show that the optimization implies only a small overhead for the WCET analysis runtime. Thus, procedure cloning is best suited to improve the analyzability of real-world applications that require a reduced analysis complexity.

In the future, we plan to incorporate a WCET-aware C compiler into the workflow by replacing the commercial compilers. This would enable an improved exploitation of procedure cloning. Currently, the compiler is not aware of any WCET information and employs heuristics to improve the average-case execution time. Within the WCET-aware compiler, cloning could be guided by data provided by the timing analyzer and primarily functions on the worst-case path could be aggressively optimized. The integration into a WCET-aware compiler also offers the opportunity to perform a trade-off between the improvements concerning the WCET and the resulting code size. Thus, to meet the restrictions on the code size defined by the system, the compiler could evaluate the functions that produce the best gain after specializing and exclusively optimize them. Furthermore, we want to study if it might be possible to perform procedure cloning virtually, i. e. to optimize the program exclusively for the WCET analysis and keep the original code structure for the generation of the binary executable. Hence, we would be able to precisely annotate the loops and avoid an code size increase in the final code.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework.

8. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2007.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[3] K. D. Cooper, M. W. Hall, and K. Kennedy. A Methodology for Procedure Cloning. *Computer Languages*, 19(2):105–117, 1993.

[4] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *ESTIMedia'06*, pages 121–126, 2006.

[5] H. Falk and M. Schwarzer. Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *ESTIMedia'06*, pages 115–120, 2006.

[6] C. Ferdinand, R. Heckmann, H. Theiling, and R. Wilhelm. Convenient User Annotations for a WCET Tool. In *WCET'03*, pages 17–20, 2003.

[7] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. In *RTSS'06*, pages 57–66, 2006.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC'01*, pages 3–14, 2001.

[9] S. Lee, J. Lee, C. Y. Park, and S. L. Min. A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor. In *SCOPES'04*, pages 244–258, 2004.

[10] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[11] S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[12] W. Zhao, D. Whalley, C. Healy, et al. WCET Code Positioning. In *RTSS'04*, pages 81–91, 2004.

A Framework for Static Analysis of VHDL Code

Marc Schlickling
Saarland University & AbsInt GmbH
schlickling@cs.uni-sb.de

Markus Pister
Saarland University & AbsInt GmbH
pister@cs.uni-sb.de

Abstract

Software in real time systems underlies strict timing constraints. These are among others hard deadlines regarding the worst-case execution time (WCET) of the application. Thus, the computation of a safe and precise WCET is a key issue[1] *for validating the behavior of safety-critical systems, e.g. the flight control system in avionics or the airbag control software in the automotive industry.*

Saarland University and AbsInt Angewandte Informatik GmbH have developed a successful approach for computing the WCET of a task. The resulting tool, called aiT, is based on the abstract interpretation [3, 4] of timing models of the processor and its periphery. Such timing models are hand-crafted and therefore error-prone. Additionally the modeling requires a hard engineering effort, so that the development process is very time consuming.

Because modern processors are synthesized from a formal hardware specification, e.g., in VHDL *or* VERILOG*, the hand-crafted timing model can be developed by manually analyzing the processor specification.*

Due to the complexity of this step, there is a need for support tools that ease the creation of analyzes on such specifications. This paper introduces the primer work on a framework for static analyzes on VHDL*.*

1 Introduction

During the last years, embedded systems have become nearly omnipresent in everyday life. Embedded processors are used in a variety of application fields: health-care technology, multimedia applications, telecommunication, automotive and avionics, weapon guidance, etc. Common characteristics of many applications are that high computation performance has to be obtained at low cost and low power consumption. Moreover many applications have safety-critical characteristics and must satisfy hard real-time constraints. This leads to an additional requirement to be respected in embedded system design: the requirement of predictable performance. It is not enough for microprocessors to yield high peak performance, but it should also be possible to statically guarantee their worst-case performance. Contemporary superscalar architectures are characterized by deep complex pipelines, often with features like out-of-order execution, branch prediction, and speculative execution which make determining the guaranteed performance of applications a difficult task [8].

The worst-case execution time analyzer *aiT* originally developed by *Saarland University* and *AbsInt Angewandte Informatik GmbH* is a tool for computing safe and precise upper bounds of the worst-case execution time (WCET) of tasks. The computation is based on the *abstract interpretation* [3, 4] of *timing models* of the processor core and its system controller [18, 19]. The tool takes the executable as input and performs several static analyzes on it. The input is transformed into an intermediate representation called *Control-Flow Representation Language* (CRL)[2] [13], on which the analyzes are based. Further details about the *aiT* tool-chain can be found in [6].

The computation of the WCET of a task mainly depends on the so called *pipeline analysis* in which the behavior of the processor pipeline and the underlying system controller are modeled. This is done by abstracting from everything that is not needed for the timing behavior of the processor pipeline. Further details about how to create a pipeline analysis can be found in [18].

As of today, these models are hand-crafted and only obtainable with a hard engineering effort. Therefore, the development of a pipeline analysis is a very time consuming and error prone process. And the complexity dramatically increases along with each new processor generation used within embedded systems[3].

A formal processor specification (usually coded in a *Hardware description language* like VHDL or VERILOG) can be very helpful in the creation of a pipeline analysis. But even then, one needs to manually analyze the specification in order to find suitable abstractions. In order to ease this we introduce a framework for static analyzes of VHDL de-

[1] besides the functional correctness of the system

[2] In our framework we use the second version of this intermediate representation, called CRL2.

[3] As of today, these embedded processors are rather similar to and featureful as modern desktop processors.

scriptions. To this end, we developed a VHDL frontend that transforms the specification into the intermediate language CRL2 mentioned above. Different static analyzers can be generated using the *Program Analyzer Generator* PAG based on a formal analysis specification.
The paper is structured as follows: Section 2 contains a description of PAG and CRL2. Section 3 then details the semantics of VHDL. In Section 4, our analysis framework is described and illustrated with an example in Section 5. Section 6 shows some experimental results. Section 7 gives an outlook on future work and Section 8 concludes.

2 Preliminaries

We use PAG to generate static program analyzers based on a control flow graph. The next two sections introduce the basics of the control flow representation language and the Program Analyzer Generator.

2.1 CRL

The Control-Flow Representation Language (CRL2) was developed to provide an intermediate format that simplifies analyzes and optimizations on a control flow graph [13].
A *control flow graph* is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Nodes in the graph represent basic blocks, i.e. straight-line pieces of code. Directed edges are used to represent jumps in the control flow.
Assuming that the control flow graph is always well-formed, the structure of CRL2 is hierarchically organized in instructions, basic blocks and routines. Thereby, the former is always completely enclosed within the latter. A sample control flow graph is given in Figure 1 showing two routines (`simul` and `environment`). The edge between the two routines indicates a call dependency, each routine call is represented through *call/return blocks* in the callee's routine. The call/return blocks ease interprocedural control flow analyzes [17] and are more or less placeholders for the branch to the called routine.
CRL2 is a very flexible language by virtue of an attribute-value concept, i.e. each element can be extended by attributes coding arbitrary information. To process a program represented by its control flow graph, PAG can be used.

2.2 Program Analyzer Generator

PAG is a powerful tool for generating program analyzers. Based on a high-level specification of a data flow problem, PAG automatically generates a program analyzer[4] which can be used in arbitrary applications [14].
The triple $(K, L, [\![]\!])$ is called a *data flow problem* for a complete lattice[5] L and a control flow graph K, if $[\![]\!] : N \rightarrow$

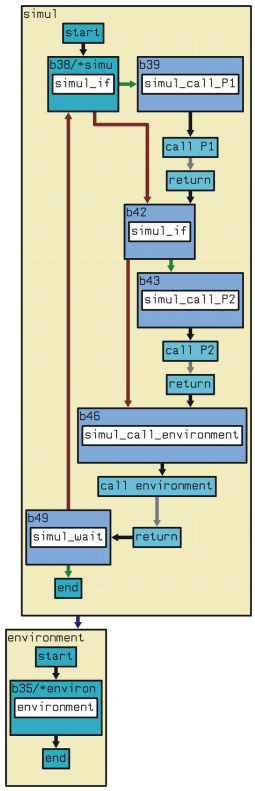


Figure 1: Sample CRL2-graph

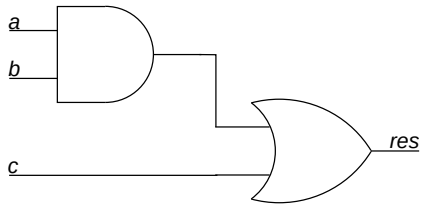
$(L \rightarrow L)$ is a function assigning functions from $L \rightarrow L$ to the nodes of K. These functions are called *transition functions* and are used for updating the data flow value during analysis.
More details on data flow problems can be found in [16], a description of the specification language for PAG in [1].

3 VHDL Semantics

VHDL is an IEEE Standard defined in IEEE 1076 [2, 12]. The focus of the language ranges from specifying circuits at wavefront level to describing large system behaviors with high-level constructs. As a result, the standard is huge. The focus of this paper only considers the *synthesizable subset* of VHDL, defined in [11].
A VHDL description of a circuit consists of an interface declaration defining the in- and output signals of the circuit and of one or more implementation(s). In VHDL, the first is called an *entity*, the second an *architecture*. Figure 2 shows a simple 3-bit counter.
The implementation is given in form of two *processes* (`P1` and `P2`). Each process executes its code, whenever one of the *signals* contained in the processes *sensitivity lists* (*clk* and *rst* for `P1`, *cnt* for `P2`) changes its value. After execution of all statements, execution suspends until another

[4] in ANSI-C

[5] A *complete lattice* is a partially ordered set in which all subsets have both a supremum and an infimum. An example for a complete lattice is the power set of a given set, ordered by inclusion. The supremum is given by the union and the infimum by the intersection of subsets.




```
entity comb_logic is
  port(a,b,c:in std_logic;res:out std_logic);
end;
architecture rtl of comb_logic is
  signal wire: std_logic;
  component and_gate is
    port(u,v:in std_logic;w:out std_logic);
  end component;
  component or_gate is
    port(x,y:in std_logic;z:out std_logic);
  end component;
begin
  and_gate port map(a,b,wire);
  or_gate port map(wire,c,res);
end;
```

Figure 3: Composition of VHDL components

```
entity counter is
  port(clk:in std_logic; rst:in std_logic;
       val:out std_logic_vector(2 downto 0));
end;
architecture rtl of counter is
  signal cnt:std_logic_vector(2 downto 0);
begin
  P1: process(clk,rst) is
    if (rst='1') then
      cnt<="000";
    elsif (rising_edge(clk)) then
      cnt<=cnt+'1';
    end if;
  end;
  P2: process(cnt) is
    val<=cnt;
  end;
end;
```

Figure 2: 3-bit counter in VHDL

change of at least one signals value. Thus, the sensitivity list of a process is an implicit wait-statement at its end.[6]

VHDL also supports component-based circuit specifications. Figure 3 gives an example for hierarchical circuit composition. Here, the combinatorial function $res = a \wedge b \vee c$ is modeled using a logical-and and a logical-or gate.

Having a hierarchical composed specification of a circuit, *elaboration* has to be performed in order to get a flat definition of it. Elaboration does all the required renaming for unifying names, wires all structural descriptions, etc. The result is one large entity consisting of a number of processes and some locally defined signals.

A VHDL process consists of a set of local *variables* that are only accessible from inside the process. By contrast, local signals can be accessed by more than one process, but only one process is allowed to drive the value of a signal.[7] Within a process, execution of statements is done sequentially.

VHDL makes a distinction between the assignments to a variable and to a signal. Assigning a value to a variable takes effect immediately (i.e., the next reference of this variable returns the newly assigned value), whereas the assignment of a value to a signal is only *scheduled* to be the future value (i.e., the next reference returns the old value). E.g., in Figure 2, the signal assignment `cnt<=cnt+'1';` schedules the next value of `cnt` to be `cnt` plus one, but the next reference `val<=cnt;` schedules the next value of `val` to be the *current* value of `cnt`. These future values take effect as soon as all processes *suspend* their execution. The semantics of a VHDL program, i.e. a set of processes, can be described as follows:

1. Execute processes until they suspend.
2. If all processes are suspended, make all scheduled signal assignments visible at once.
3. If there is a process being sensitive on a signal having changed its value, resume this process and go to step 1.
4. Otherwise, an external signal must change its value (e.g., the clock signal). If this happens, resume all processes waiting for this signal and go to step 1.

Thus, the semantics of VHDL can be seen as a two-level semantics: sequential process execution at its first, signal update and process revocation at its second level.

4 A VHDL analysis framework

As mentioned in the introduction, we present here a framework for statically analyzing VHDL code using *abstract interpretation*. The structure and data flow of our framework is illustrated in Figure 4. As input, we have the VHDL model that we want to analyze and a PAG specification for a static analysis. We developed a tool

[6] In VHDL, the use of explicit wait-statements and sensitivity lists is exclusive. We assume, that the only place within a process, where wait-statements may occur, is at the end of the body of a process.

[7] In full VHDL, *resolution functions* can be used for value computation of signals being driven by two or more processes.

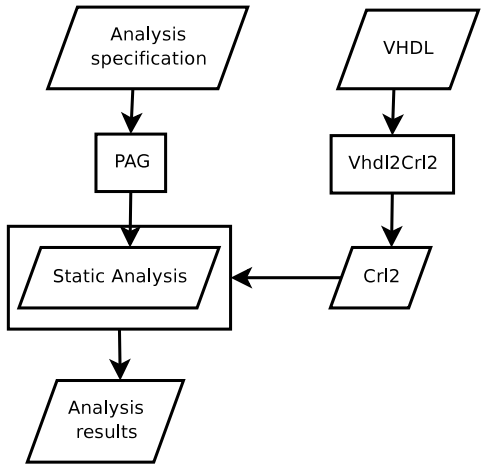


Figure 4: Structure of VHDL analysis framework

VHDL element	CRL2 element
Process, Function, Procedure, Concurrent signal assignment, Concurrent procedure call, Loop	Routine
Function calls Procedure calls	Routine calls Routine calls
Sequential statement	Instruction

Table 1: Mapping VHDL to CRL2

called `Vhdl2Crl2` that transforms the VHDL into semantically equivalent CRL2 constructs. From the analysis specification, PAG generates a static analyzer that works on the CRL2 description. And at the end, the analysis emits its results. The mentioned components in our framework are now detailed in the following subsections.

4.1 Mapping VHDL to CRL2

In order to express a VHDL description in semantically equivalent CRL2 constructs, we need to give a mapping from VHDL components to CRL2 components. Table 1 shows this mapping. Processes are transformed into routines as well as functions, procedures, concurrent signal assignments, concurrent procedure calls and loops. The transformation of loops to routines roughly means, that the whole loop body is moved into a newly created routine and the original location of the loop is replaced by a call to the new loop routine. This improves the quality of static analyzes of loops. More details on this so called *Loop transformation* can be found in [15].

The correspondence of function/procedure calls to routine calls as well as the mapping of sequential statements to instructions is rather intuitive.

4.2 VHDL as a sequential program

Regarding the special semantics of VHDL (cf. Section 3), we need to express the input VHDL description as a sequential program whose control flow is represented by CRL2. The reason for this is that we want to generate the analyzer itself from a concise PAG specification, where PAG is an analyzer generator for programs (cf. Section 2).

As mentioned in Section 3, variables in VHDL are process-local and processes run in parallel. Additionally signal assignments only take effect after all processes have finished their execution. These semantics directly induce that there are no side effects between the different VHDL processes. So, we can serialize their execution without changing the semantics of the whole model.

In order to formulate a VHDL description as a sequential program, we just need to choose an arbitrary execution order among the processes and iteratively execute them in this order. The program then consists of a routine, let's call it `simul` (cf. Figure 1), whose body contains routine calls, where each called routine represents one of the former VHDL processes. Additionally each such routine call for an original process it guarded with a conditional statement that evaluates the sensitivity list of the process. If at least one of the signals in the sensitivity list has changed, the call is taken. The result of such a transformation from a VHDL model into a sequential program is shown in Figure 1. Here, you can see the `simul` routine, containing routine calls for each VHDL process, namely for `P1` and `P2`. The basic blocks *b38* and *b42* contain an instruction `simul_if` that represents the guards for the sensitivity list evaluation. The basic block *b49* containing the instruction `simul_wait` is the so called synchronization point. Here, the signal assignments take effect and we can decide whether a signal value has changed compared with the previous iteration.

In Figure 1 there is another call instruction not mentioned so far, the `simul_call_environment` that calls the routine `environment`. We need this routine for analyzing open systems, which roughly are systems that have external input signals. These signals have to be set somewhere in the environment of the system modeled by the VHDL description and they drive the behavior of the whole system. One intuitive example for such an external signal is the `reset` signal.

4.3 Modeling the clock

If we want to analyze synchronous designs, i.e. systems that are synchronized with either the rising or the falling edge of the clock signal, we need to model this signal somehow. Unfortunately the synthesizable subset of the VHDL standard [11] does not enable us to model a clock signal[8].

[8] In synthesizable VHDL, there are two main restrictions: a process can not be sensitive on a signal it drives and there is no possibility to wait for a timeout. Thus, there is no construct left for modeling the frequent change of a clock signal.

Despite this, we can simulate the clock easily by introducing a new routine, called `clock`, that calls the routine `simul` twice. Before one call the clock signal is set to one and before the other call the clock signal is set to zero. This makes our approach rather flexible as we can analyze synchronous designs as well as asynchronous ones.

5 Example: Constant Propagation

This sections describes, how the framework introduced in Section 4 can be used to model a constant propagation analysis on VHDL.

A *constant propagation analysis* determines for each program point, i.e. each statement, if a signal or variable has a constant value, when execution reaches that point. To model this, we introduce a mapping from identifier names to their corresponding value and extend it by the usual bottom and top elements to denote not yet considered program points and unknown values respectively.

$$F \equiv (\mathit{identifier} \rightarrow (\mathit{value} \cup \top)) \cup \bot$$

As stated in Section 3, VHDL differs between signal and variable assignments. Thus, the domain of the data flow problem for constant propagation analysis has to cover *current* and *future* values of the identifiers used. Furthermore, to evaluate the condition of process guards (`simul_if`, see Section 4.2), it is necessary to decide, whether a signal has changed its value or not. Therefore, the domain has to be extended by the *old* values of signals. Thus, the domain *dfi* for the constant propagation analysis is:

$$\mathit{dfi} \equiv F \times F \times F$$

The transition functions for updating an incoming data flow value $\mathit{dfi}_{pre} = (\mathit{cur}_{pre}, \mathit{fut}_{pre}, \mathit{old}_{pre})$ to the output value dfi_{post} for the different nodes can be directly defined as follows:

- assignment node
  The data flow value for an assignment can be computed from the incoming value in case of a variable assignment by updating the current and the future value of dfi_{pre} with the newly assigned value or $\top$, if this value is statically not computable. In case of a signal assignment, only the future value has to be updated.
- `simul_if` node
  The guard encapsulates the sensitivity list of a process and is responsible for the repeated execution of it. A process is only re-executed, if one of the signals in its sensitivity list changes its value. This can be checked by comparing cur_{pre} with old_{pre}. Based on this result, the data flow value is propagated into the process or not.
- `sync` node
  At this node, all scheduled signal assignments take effect and are made visible at once. The new data flow value is computed by copying the future values to the current one and the current ones to the old ones.

  $$\mathit{dfi}_{post} = (\mathit{fut}_{pre}, \mathit{fut}_{pre}, \mathit{cur}_{pre})$$

- environment node
  Writing a special rule for this node allows us to analyze the VHDL code with respect to special system criteria. E.g., if we want to analyze the reset behavior of the code displayed in Figure 2, we introduce a rule $\mathit{cur}_{post} = \mathit{cur}_{pre} \backslash [rst \rightarrow 0]$ and $\mathit{fut}_{post} = \mathit{fut}_{pre} \backslash [rst \rightarrow 1]$ always setting the current and future value of `rst` to 0 and 1 respectively. This yields to the perception, that `cnt` and therewith `val` have constant values during reset.

Using PAG and the rules above yields in a constant propagation analysis on VHDL. The results can now be used for further analyzes and transformations of the VHDL code as described in Section 7.

6 Experiments

Despite the example given in the previous section, we successfully[9] tested our implementation with the complete VHDL specification of the Leon2 processor core [7]. The Leon2 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture with a 5-stage pipelined integer unit, data and instruction cache, hardware multiply, divide and MAC units. The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. The VHDL specification of the Leon2 consists of more than 80 modules containing about 75.000 lines of code.

7 Outlook

The analysis framework described in this paper is only one part of a much bigger task: The semi-automatically derivation of timing analyzers from a formal processor specification.

With the work here, we are able to read VHDL specifications and to perform static analyzes like constant propagation on it. This eases the task of finding suitable abstractions for a processor model, i.e. cropping the model to the parts that are only relevant for the timing behavior.

As noted earlier, a VHDL design is too large to be used directly in timing analysis, thus we have to throw away things not influencing the timing. This can be achieved by slicing backwards from the place, where instructions leave the pipeline, i.e. finish execution. Only external signals and variables or those being assigned to in the slice can influence timing. Signals and variables used (i.e. read) but not assigned in the slice do not change their values (i.e. have fixed values) and can be omitted.

[9] Successfully here means, that we are able to create static analyzers based on analysis specifications.

Even after this removal, the model may be too large for timing analysis. Thus, we approximate concrete components by abstract ones (see [6] for more details) which are smaller in size. An example can be found in [5].
For this, we want to develop support tools to incorporate transformations on the VHDL model based on the analysis results semi-automatically in order to derive abstract models.
From an abstract processor model, we need to generate a C-code analysis fitting into the tool chain of the *aiT* tool.
At the end of the story, we want to derive a timing analyzer from a formal hardware specification. This not only speeds up the task a creating such a timing analysis but additionally the generated analyzer is already validated due to the derivation from the hardware specification.

8 Conclusions

Safety-critical applications as for example the flight control software in avionics or the airbag control software in the automotive industry are underlying hard real-time constraints. Therefore the computation of the worst-case execution time (WCET) is the key issue for guaranteeing that a system satisfies its timing boundaries.
The growing complexity of modern processor architectures used within such safety-critical systems complicates the task of creating a sound timing analysis. Currently, these analyzes are based on hand-crafted abstract processor models. But this is a very time consuming and error-prone process.
To ease the task of finding suitable abstractions, we introduced a framework for static analyzes of formal processor descriptions in VHDL. By transforming the VHDL model into a sequential program, we can generate static analyzers from a concise specification using the *Program Analyzer Generator* [14]. Our framework is very flexible because we can analyze open designs as well as closed ones, i.e. systems that does or does not depend on external driven signals respectively.
By using CRL2 as the intermediate representation, we can combine several analyzes. This means that an analysis can use the result of another one because the results are annotated as attributes in our intermediate representation.
To illustrate the practicability of our framework, we showed how to create a constant propagation analysis on a VHDL description of a 3-bit counter.
In [9, 10] Charles Hymans gives a design for static analysis of VHDL that uses abstract interpretation. Despite this and to the best of our knowledge, our work presented here is the first work concerning a framework for generating static analyzers on VHDL code. Our results are currently going to be used for the semi-automatically derivation of timing analyzers from a formal processor description in VHDL.

Acknowledgments

This work was partly 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) and by EU network of excellence IST-004527 ARTIST2 on Embedded Systems Design.

References

[1] AbsInt Angewandte Informatik GmbH. *The Program Analyzer Generator User's Manual*, 2002.
[2] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
[4] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2-3):103–179, 1992.
[5] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, 1997.
[6] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation. Theory and Practice.*, volume 4444 of *LNCS*, pages 12–52. Springer, 2007.
[7] J. Gaisler. *Leon2 Processor User's Manual - Version 1.0.30*, July 2005.
[8] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.
[9] C. Hymans. Checking safety properties of behavioral vhdl descriptions by abstract interpretation. In *SAS*, pages 444–460, London, UK, 2002. Springer.
[10] C. Hymans. Design and implementation of an abstract interpreter for vhdl. In D. Geist and E. Tronci, editors, *CHARME*, volume 2860 of *LNCS*. Springer, 2003.
[11] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076.6 1999 VHDL Register Transfer Level Synthesis*, 1999.
[12] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076 2000 VHDL Language Reference Manual*, 2000.
[13] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, 1998.
[14] F. Martin. *Generating Program Analyzers*. PhD thesis, Saarland University, 1999.
[15] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In K. Koskimies, editor, *CC*, volume 1383 of *LNCS*. Springer, 1998.
[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
[17] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, 1981.
[18] S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
[19] S. Thesing. Modeling a System Controller for Timing Analysis. In *EMSOFT*, pages 292–300, 2006.

Towards Symbolic State Traversal for Efficient WCET Analysis of Abstract Pipeline and Cache Models *

Stephan Wilhelm
AbsInt GmbH and Saarland University
Saarbrücken, Germany
sw@absint.com

Björn Wachter
Saarland University
Saarbrücken, Germany
bwachter@cs.uni-sb.de

Abstract

Static program analysis is a proven approach for obtaining safe and tight upper bounds on the worst-case execution time (WCET) of program tasks. It requires an analysis on the microarchitectural level, most notably pipeline and cache analysis. In our approach, the integrated pipeline and cache analysis operates on sets of possible abstract hardware states. Due to the growth of CPU complexity and the existence of timing anomalies, the analysis must handle an increasing number of possible abstract states for each program point. Symbolic methods have been proposed as a way to reduce memory consumption and improve runtime in order to keep pace with the growing hardware complexity. This paper presents the advances made since the original proposal and discusses a compact representation of abstract caches for integration with symbolic pipeline analysis.

1. Introduction

Finding the worst-case execution time (WCET) for all tasks of a software is an important requirement in the design of hard real-time systems. A proven approach for obtaining tight upper bounds of the WCET, based on *abstract interpretation* (AI), has been presented in [9]. It employs several semantics-based static program analyses on the assembly level control flow graph (CFG) of the input program. First, the *value analysis* computes possible register contents for each program point in order to determine the address ranges for instructions accessing memory. Then, an integrated *pipeline and cache analysis*, operating on safe approximations of the possible pipeline and cache states, computes a WCET bound for each basic block of the CFG. Safe approximation means, that the analysis might only consider too many states, i. e. the WCET state is always included. The correctness of this approach has been proven [8] [15]. Finally, a *path analysis* computes the global worst-case path using the WCET bounds for basic blocks determined by the pipeline and cache analysis [14]. The AI approach has been used very successfully for various complex, real-life architectures [16] [13].

Unfortunately, CPUs using modern techniques for reducing the average execution time, such as caches, pipelined execution, branch prediction, speculative execution, and out-of-order execution, are often subject to *timing anomalies*. A timing anomaly is a local worst-case behavior, e. g. a cache miss, that does not contribute to the global worst-case [11]. As a consequence, abstract interpretation of the pipeline behavior must consider a large number of possible abstract pipeline states for each program point. This problem is also known as *state explosion*. In certain cases, the analysis can even become infeasible because of the increase in memory consumption and computation time [15].

Measurement-based approaches for computing the WCET avoid the high cost of microarchitectural modeling and are therefore not affected by the problem of state explosion. However, measurement-based approaches are often not suitable for safety critical applications since an underestimation of the WCET cannot be excluded [2]. Furthermore, using measurement-based methods for complex architectures can lead to a high overestimation of the WCET [7].

Symbolic methods have been proposed recently as a way to handle sets of abstract pipeline states efficiently, reducing memory consumption and improving runtime, in order to keep pace with the growing hardware complexity [17]. In the past, such methods have been used successfully for similar problems in model checking [5]. This paper presents an extension of the AI approach for pipeline analysis, using a symbolic representation of abstract pipeline states.

*This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/) and by the Transregional Collaborative Research Center 14 AVACS (http://www.avacs.org/).

Our Contribution. We propose an improved algorithm for symbolic state traversal of abstract pipeline models that overcomes limitations of the algorithm presented in [17]. Furthermore, we discuss a compact encoding of abstract caches that admits an integration of pipeline and cache analysis.

The paper is organized as follows. Section 2 briefly introduces the required terminology and section 3 gives an informal overview of the abstraction techniques used for deriving abstract pipeline models. Then, we review the algorithm for computing state transitions for sets of abstract pipeline states, using a small example (section 4). Using the same example, section 5 shows that the algorithm does not handle all cases correctly and proposes a solution for this problem. Section 6 discusses the performance of the approach and section 7 presents the current state of our implementation and gives first performance numbers. Finally, section 8 discusses the integration with a cache analysis.

2. Background

Given a finite state machine (FSM) with a set of states Q, a set of input values I, and a transition relation T, each set of FSM states $A \subseteq Q$ can be associated to its *characteristic function* $\mathbf{A} : Q \to \{0,1\}$; $\mathbf{A}(x) = 1 \Leftrightarrow x \in A$. In the same way, the *transition relation* T can be associated to the function $\mathbf{T} : Q \times I \times Q \to \{0,1\}$; $\mathbf{T}(x,i,y) = 1 \Leftrightarrow (x,i,y) \in T$. It is common practice to represent FSM state sets and the FSM transition relation by their characteristic functions encoded as *binary decision diagrams* (BDDs). BDDs are usually more compact than explicit representations and efficient implementations of useful operations such as negation, conjunction and existential quantification exist [4].

Given a set of FSM states $A \subseteq Q$, the *image* of A, $\mathrm{Img}(A) \subseteq Q$, is the set of states that is reachable from A under T. Image computation is the core operation of symbolic model checking algorithms and can be efficiently implemented using BDDs [12].

Pipeline analysis is a static program analysis which performs a fixed point iteration on the domain of abstract pipeline states. The least fixed point (LFP) is the solution to the data flow problem containing all states that are reachable for a given program point including the WCET state. The result is a maximum number of cycles for each basic block.

Symbolic pipeline analysis is an implementation of pipeline analysis using BDDs for representing sets of abstract pipeline states (an abstract pipeline model is an FSM). In addition to the standard BDD operations provided by any BDD library, it uses the image computation engine from a model checker for efficiently computing the reachable abstract pipeline states.

3. Abstract Pipeline Models

Implementation of an abstract pipeline model requires detailed knowledge about the internal working and timing behavior of a CPU. This knowledge can be obtained from written documentation or hardware traces of bus signals or from the original Verilog or VHDL implementation (although the latter is often not available). Using any of this information, an abstract pipeline model can be derived by

1. Omission of timing-irrelevant implementation details.
2. Omission of data paths.
   Operations on the register file are already handled by the value analysis. Possible register values are therefore available during pipeline analysis without modeling register file and ALU.
3. Use of instruction addresses in all pipeline stages.

The last two points require a detailed discussion because they are crucial for understanding the difference between model checking and pipeline analysis. Model checking explores the state space of a model in order to prove or reject a statement in temporal logic. All information is contained in the model and state space exploration is guided by the transition relation and by the logic statement. In contrast, pipeline analysis explores the model's state space guided by the transition relation and the structure of a program and by additional information from other analyses. Thus, the model does *not* contain all information because we have delegated some of it, e. g. to the value analysis. In order to obtain this information during the analysis, we need to establish a relationship between abstract states and program points. This is achieved by the use of instruction addresses in all pipeline stages (see item 3 above). The same relationship can be used to annotate the analysis result, i. e. the WCET for each basic block.

Reading the delegated information during state transition is trivial for implementations operating on an explicit representation of abstract states, because the transition is computed individually for each abstract state [15]. On the other hand, symbolic pipeline analysis gains efficiency by using BDD operations on sets of abstract states. Explicitly extracting and recoding single states has to be avoided or minimized. An algorithm for computing state transitions on sets using image computation and encoding the delegated information using BDD operations has been presented in [17]. In the next section, we will illustrate the key ideas using a simplified abstract pipeline model.

4. An Illustrated Example

Figure 1 shows a simplified extract from our abstract pipeline model for a subset of the Infineon Tricore [18]. The

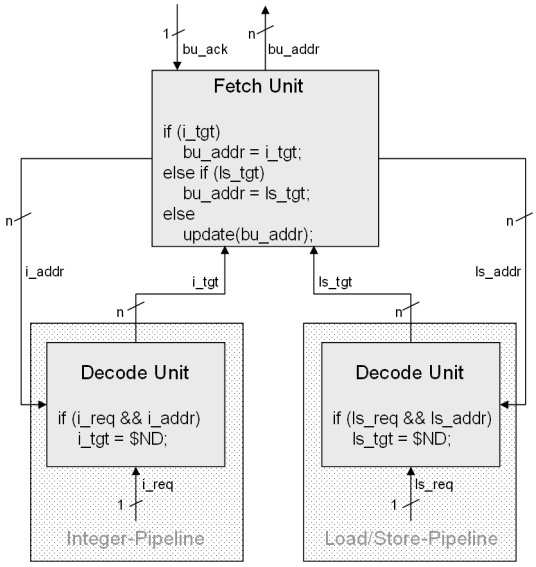


Figure 1: Extract from Tricore model.

architecture features two pipelines, called *Integer-Pipeline* (IP) and *Load/Store-Pipeline* (LSP), which share the same fetch unit. Each pipeline has its dedicated decode unit. [1] The state of each unit is held in a few variables which are updated in each cycle. The Verilog code below the unit's name shows the update for three selected variables. The current value of each variable is communicated to other units using `1`- or `n`-bit wide wires, where `n` is the number of bits used for encoding instruction addresses.

Let us examine what happens if we analyze this subset of our abstract model. The fetch unit sends instruction addresses to the bus unit via `bu_adr` which returns the signal `bu_ack` when the instruction data arrives. The fetch unit controls an 8 byte prefetch buffer (not shown in the example) and dispatches its contents to the decode units. The fetch unit sends only instruction addresses via `i_addr` and `ls_addr` instead of sending instruction data (remember that data paths have been removed). The decode units are responsible for detecting structural pipeline hazards (not shown here) and for computing targets of control-flow instructions. Target addresses are stored in the `i_tgt` and `ls_tgt` buffers where the special value `0` indicates that the buffer does not contain a valid target. Whenever the fetch unit is about to request a new address from the bus unit, it checks whether any of the two buffers contains a valid address and in that case it overwrites `bu_addr` with that address and redirects the next fetch to the branch, return or call target. Otherwise, the new value of `bu_addr` is calculated by the function `update` depending on the state of the prefetch buffer.

[1] The third pipeline for handling zero-overhead loops shares its decode stage with the LSP.

```
0xd4000056  mov d15, +21649
0xd400005a  j 0xd4000068
```

Figure 2: Tricore assembly.

Pipeline analysis starts from the initial state where `bu_addr` contains the start address and `i_tgt` and `ls_tgt` are set to zero. The reachable pipeline states are computed by repeated image computation, but at certain points we require some of the delegated information (from now on, we will refer to such cases as *external requests*). E. g. in a state q where the condition `(i_req && i_addr)` holds, we require the possible control-flow targets for the instruction at address `i_addr` in order to update the variable `i_tgt`. The update rule [2] for this variable, `i_tgt = $ND`, states that `i_tgt` may adopt any possible value in the next cycle (`$ND` stands for non-deterministic values). Thus, the image of q is a set of 2^n states that differ only in the value of `i_tgt`.

Consider the assembly code of figure 2 and let us assume that the value of `i_addr` in state $q \in Q$ is `0xd400005a`. A lookup of this address from the assembly program shows that it is an unconditional jump to address `0xd4000068`. The key idea presented in [17] is to partition a set of abstract states according to different external requests. Computing the image of each partition yields all possible successor states for that request, which can be restricted using the external information. The set of successor states for the next analysis cycle is the disjunction of the restricted images for each partition.

5. Concurrent External Requests

Consider again the assembly code of figure 2. The Tricore fetch unit can issue such a pair of instructions concurrently by assigning the move instruction to the IP and the unconditional jump to the LSP. If the signals `i_req` and `ls_req` are active during the next cycle, then both conditions `(i_req && i_addr)` and `(ls_req && ls_addr)` hold. This means that we have two external requests in the same state $r \in Q$ and `Img`(r) produces 2^{2n} successor states, because *both* target buffers may adopt any value. The original algorithm as proposed in [17] handles each external request individually and therefore fails to create a correct restriction for this case. Thus, if restrict$(A, \mathtt{v})$ denotes the restriction of variable v in the subset $A \subseteq Q$, then the algorithm effectively computes

$$\text{restrict}(\text{Img}(\{r\}), \mathtt{i\_tgt}) \cup \text{restrict}(\text{Img}(\{r\}), \mathtt{ls\_tgt})$$

This computation yields $2 \cdot 2^n$ states instead of the single state where `i_tgt` equals `0` (the move has no branch target) and `ls_tgt` equals `0xd4000068`.

[2] Depicted in the left decode unit in figure 1.

We therefore propose a slightly different algorithm for partitioning the set of abstract pipeline states, in order to simplify finding all restrictions for concurrent external requests. It involves the computation of all *cofactors* of the variables controlling external requests. Computing a cofactor means restricting a binary variable to either 1 or 0. We define an ordering on the decision variables for external requests, e. g.

$$\begin{array}{ccccccc} \texttt{i\_req} & \underset{1}{\rightarrow} & \texttt{i\_addr[n]} & \rightarrow & \ldots & \texttt{i\_addr[0]} \\ \downarrow & & & & & & \\ \texttt{ls\_req} & \underset{1}{\rightarrow} & \texttt{ls\_addr[n]} & \rightarrow & \ldots & \texttt{ls\_addr[0]} \end{array}$$

All decision variables for a single external request appear in a single line and their ordering is indicated by arrows. Using this ordering, we compute a tree of cofactors as shown in figure 3. The leaves of the tree are the required partitions for external requests. In some cases, we do not need to consider the cofactors of all variables in a line. E. g. we skip cofactoring the variables `i_addr[n]` ... `i_addr[0]` for cofactors where `i_req` $= 0$ because the expression

$$\texttt{i\_req} \wedge (\texttt{i\_addr[n]} \vee \ldots \vee \texttt{i\_addr[0]})$$

evaluates to false for all assignments of `i_addr[n]` ... `i_addr[0]`. This shortcut is indicated by the 1 below the ordering arrow between `i_req` and `i_addr[n]`. Shortcuts can be found easily because of the regular structure of external requests. They usually rely on the state of very few signals (like `i_req`) and the address part must be different from 0 (invalid address). The partitions depicted in figure 3 are

- (A) States without external requests.
- (B) States requesting `i_tgt`.
- (C, D) States requesting `i_tgt` and `ls_tgt`.

The states (C) and (D) differ in the instruction address `ls_addr` for the external request of `ls_tgt`. In practice, most of the possible cofactors will be empty, which means that it suffices to consider a fraction of the possible paths through the partitioning tree. The efficiency of the partitioning can be further improved by choosing a variable ordering that allows for the definition of many early shortcuts. Furthermore, the use of shortcuts avoids the partitioning of states with different addresses but without active external requests, e. g. for `i_addr` if `i_req` $= 0$. A tree walk from the root to a leaf defines an assignment of all decision variables for all active external requests in that partition. Using this information, we can lookup the results for all active external requests of a partition and restrict the possible successor states as described in section 4.

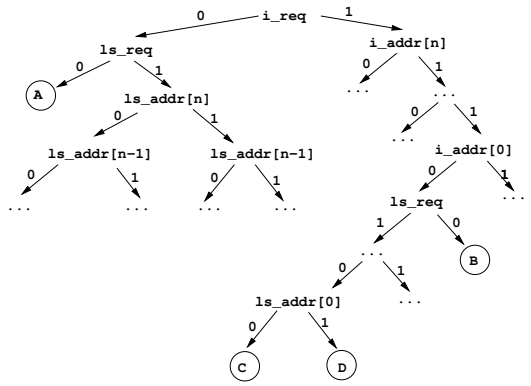


Figure 3: Partitioning tree.

6. Performance Considerations

The performance of BDD-based algorithms depends on the number of BDD variables and on their ordering. The number of BDD variables is equivalent to the number of bits needed for all variables in the abstract model. Therefore, introducing instruction addresses into all pipeline stages creates a serious problem if we use all 31 bits needed for addressing the whole address space (for Tricore, we can omit 1 bit because of alignment restrictions). We can significantly reduce the number of bits by enumerating all instructions used in the program. For external requests, we translate the numbers back using a simple table lookup. The same method can be used for compactly encoding data addresses.

The problem of finding a good variable ordering for obtaining small BDDs has been studied extensively and many heuristics exist [10]. However, finding a good ordering requires careful engineering and knowledge about the implementation of a concrete abstract model. Thus, it must be determined for each abstract pipeline model.

The number of partitions generated by our traversal algorithm depends on the number and placement of external requests in the abstract pipeline model. A large number of external requests can lead to *fragmentation* of the symbolic representation, e. g. if each state issues a different external request, we have to separate all states into singular partitions. However, we believe that the worst case is unlikely in practice and that efficient designs can be found.

7. Implementation and Experimental Results

The presented approach for symbolic pipeline analysis has been implemented in the *aiT*-framework [1], using the *VIS* [3] model checker for image computation and BDD operations. An abstract pipeline model for a subset of the Infineon Tricore has been implemented in Verilog. It comprises the shared fetch unit (fully implemented) and simplified im-

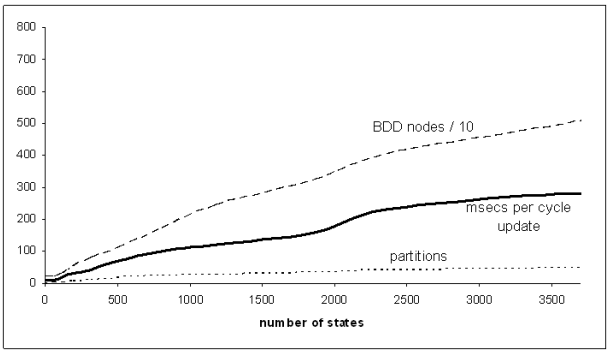


Figure 4: Cycle update times compared to number of states.

plementations of the two main pipelines (altogether about 500 lines of Verilog). The Verilog specification is translated to a netlist for VIS, using the *vl2mv* compiler [6]. The interface to aiT's program analysis framework, including the handling of external requests, is implemented in C++. The analysis has been tested successfully on several small benchmarks, e. g. dhrystone.

In order to assess the efficiency of our approach, we have increased the number of abstract states considered by the analysis by assuming different latencies for instruction fetches. This is similar to real-world problems since state explosion is often caused by uncertain information about the timing of memory accesses (e. g. cache-hit or miss). Using 63 possible latencies for each instruction fetch, we obtained the results presented in figure 4 by analyzing a program for calculating prime numbers. The solid bold curve indicates the time required for computing a cycle update for all possible abstract states at a program point. The dotted curve below shows the number of partitions generated by our algorithm and the dashed curve shows the number of BDD nodes (divided by 10). Considering that the representation of a single state requires 192 BDD nodes and approximately 10 msecs per cycle update, figure 4 shows that memory consumption (number of required BDD nodes) and computation time for cycle updates only grow slowly with the number of states. We expect that the symbolic approach will outperform an explicit implementation, as soon as the number of states reaches a break-even point.

8. Integrating Cache Analysis

We have mentioned that the AI approach uses an *integrated* cache and pipeline analysis (see section 1). In fact, the analysis operates on tuples of abstract pipeline- and cache states because of their interdependence, i. e. the order of memory accesses depends on pipeline effects and the timing of memory accesses depends on the cache state. We propose a symbolic implementation of an integrated pipeline and cache analysis such that the extra number of BDD variables for representing abstract caches remains small. Let us start by recalling some notions of cache analysis as defined in [8].

Abstract Cache Model. We deal with A-way associative caches. Let A denote the associativity of the cache. We denote by $\mathcal{M}$ the set of memory locations that are mapped into the cache. Let s be a cache set and $\mathcal{S}$ the set of cache sets. Each memory location falls into a particular cache set. The set $\mathcal{M}$ decomposes into disjoint sets $\mathcal{M} = \dot{\bigcup}_{s\in\mathcal{S}}\mathcal{M}_s$ where $\mathcal{M}_s$ are the memory locations falling into set s. Abstract caches are typically based on the concept of age. A memory location is either not in the cache, i.e. has age $\perp$, or it is in one of the A many lines of its cache set, i.e. has age k where $k \in \{1, \ldots, A\}$. The age of a memory location is an element of $\mathcal{A} = \{1, \ldots, A, \perp\}$. An abstract state of a cache set is a total function $[\mathcal{M}_s \to \mathcal{A}]$. An abstract cache state is the collection of the states of its sets:

$$\widehat{Cache} = \biguplus_{s\in\mathcal{S}} [\mathcal{M}_s \to \mathcal{A}]$$

Symbolic Representation. We exploit knowledge about the program and approximations to arrive at a compact symbolic encoding. As noted earlier, we know the memory locations that will be accessed by the program in advance, as they can be determined by value analysis. In general, the set of accessed memory locations $\mathcal{M}$ is significantly smaller than the full address space and depends on the program under analysis. Note that we encode functions that keep track of the age of particular memory locations. A straightforward encoding would be to extend the state vector by $|\mathcal{M}|$ many entries, one for each memory location, that record the age of each memory location. An advantage of this encoding is that it can be implemented using off-the-shelf image computation. However, the number of required state bits would be too high, as e. g. a data cache typically may contain tens of thousands of memory locations.

The problem with the naive encoding is that it unrolls the domain of the functions that represent cache state. In our encoding, we exploit that BDDs can encode functions directly and without unrolling, so that the number of required boolean variables is logarithmic in the number of memory locations rather than linear. For simplicity, we can assume that only one cache state is stored per pipeline state. States that agree in terms of pipeline state can be merged to one state by using the join operator for abstract caches. The symbolic domain consists of partial functions from pipeline to cache state:

$$\widehat{Pipe} \hookrightarrow \widehat{Cache}$$

An element of this domain represents a set of abstract states and is stored in a single BDD. The BDD contains the

boolean variables of the pipeline state plus boolean variables that address a particular cache set, a memory location, and age, respectively. One can use binary encoding for the set of cache sets, the sets of memory locations and ages, respectively. One obtains the following number of required boolean variables in the BDD:

$$\|\widehat{Pipe}\| + \underbrace{\lceil \log_2 \mathcal{S} \rceil + \lceil \log_2 \max_{s \in \mathcal{S}} |\mathcal{M}_s| \rceil + \lceil \log_2 |\mathcal{A}| \rceil}_{\|\widehat{Cache}\|} \quad (1)$$

where $\|\widehat{Pipe}\|$ denotes the number of bits required for pipeline state.

As mentioned earlier, BDDs typically represent characteristic functions in symbolic state traversal [5]. This is not the case for the proposed symbolic representation and therefore a modified image computation algorithm is required. However, space precludes us from discussing this algorithm.

Let us briefly assess the compactness of the representation on a real-life example, a task set obtained from an industry project. The task set was compiled for a PowerPC 755 processor featuring a 2-way associative cache with 128 sets and 32 bytes line size. We obtained (an overapproximation of) the set of accessed memory locations $\mathcal{M}$ by value analysis. The maximum number of memory locations that fall into a cache set is 94 in this example. Overall there are about ten thousand memory locations. Using formula 1, we see that the extra number of bits required for the data cache is 16 ($|\mathcal{S}| = 128$, $\max_{s \in \mathcal{S}} |\mathcal{M}_s| = 94$,$|\mathcal{A}| = 3$). In the example, the instruction cache can potentially contain only about a thousand locations. Thus we only need one more bit to switch between instruction and data cache in the representation (analogously to how we use bits to address different sets in a cache). The overall number of bits required for the cache is thus 17.

9. Conclusion

We have shown that symbolic pipeline analysis is a static program analysis which uses the same abstractions as proposed in [15]. The use of a symbolic representation for abstract pipeline states requires algorithms for efficiently encoding delegated information when computing state transitions. We have illustrated our solution using the notion of external requests (sections 4 and 5) and described its effects on the performance of the analysis.

Currently, our implementation only supports a pipeline analysis. We have outlined ideas that admit an integrated cache and pipeline analysis without significantly increasing the number of state bits compared to stand-alone pipeline analysis (section 8). In the future, we will implement the integrated analysis.

References

[1] http://www.absint.com/aiT/.

[2] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel. WCET Coverage for Pipelines. Technical report, 2006.

[3] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In *CAV*, pages 428–432, 1996.

[4] R. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. IEEE Comp. Soc. Press, 1990.

[6] S.-T. Cheng. Compiling Verilog into Automata. Technical report, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[7] A. Colin and S. M. Petters. Experimental Evaluation of Code Properties for WCET Analysis. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 190, Washington, DC, USA, 2003. IEEE Computer Society.

[8] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[10] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Variable Ordering for FSM Traversal. In *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[11] T. Lundquist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[12] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification, 1995.

[13] J. Souyris, E. Le Pavec, G. Himbert, V. Jgu, G. Borios, and R. Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[14] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, 2002.

[15] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[16] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.

[17] S. Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005.

[18] S. Zarnescu. *TriCore Pipeline Behaviour & Instruction Execution Timing*. Infineon Technologies, 2001.

Finding DU-Paths for Testing of Multi-Tasking Real-Time Systems using WCET Analysis

Daniel Sundmark, Anders Pettersson, Christer Sandberg, Andreas Ermedahl, and Henrik Thane
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{daniel.sundmark,anders.pettersson,christer.sandberg,andreas.ermedahl,henrik.thane}@mdh.se

Abstract

Memory corruption is one of the most common software failures. For sequential software and multi-tasking software with synchronized data accesses, it has been shown that program faults causing memory corruption can be detected by analyzing the relations between defines and uses of variables (DU-based testing). However, such methods are insufficient in preemptive systems, since they lack the ability to detect inter-task shared variable dependencies. In this paper, we propose the use of a system level shared variable DU analysis of preemptive multi-tasking real-time software. By deriving temporal attributes of each access to shared data using WCET analysis, and combining this information with the real-time schedule information, our method also detects inter-task shared variable dependencies. The paper also describes how we extended the SWEET tool to derive these temporal attributes.

1 Introduction

Software complexity, and especially that of embedded real-time systems, is rapidly increasing. Consequently, the task of finding faults is getting more difficult. Among the most common software failures is memory corruption, e.g., out-of-bound writes, pointer failures, and usage of uninitialized variables. For multi-tasking systems, failures also encompass non-synchronized reads and writes, and non-reentrance failures. There exist several methods that address these problems, typically in terms of static define and use analysis (DU analysis), or DU-based testing methods. However, the majority of these methods only address systems with a single thread of non-preemptive execution [7, 9, 8], or multi-tasking systems with task interference restricted to synchronous interference [3, 4].

In our previous work we have addressed testing of multi-tasking real-time systems where input and output from a task is given and produced at the beginning and at the end of the task's execution respectively [15]. We relaxed the model to encompass systems where task communication could be performed within semaphore guarded critical sections [14], and later also by non-synchronized shared variables anywhere in the execution of the tasks [13]. In this paper, we extend previous results and show how to derive all DU-paths from a preemptive real-time system using WCET analysis.

```
1.  a:=b+4
2.  if expression is true then
3.      result:=a+1
4.  else then
5.      result:=a*2
```

Figure 1. Example of defs and uses.

2 Background

Classic data-flow unit testing, for single thread execution programs, tests read and write accesses to program variables [12]. Data-flow is identified in terms of definitions and uses of data, where a *definition* is an assignment of a value to a variable. A *use* is an action of reading a variable or container. One classic DU relation is the *DU-path*. A definition d (write) and a use u (read) of variable x constitutes a DU-path (d,u) if and only if there exists a control-flow path p from d to u, such that p contains no other definitions of x. E.g., in Figure 1, uses are enclosed in a selection statement, yielding the following DU-paths: $\{(\mathtt{a}^1, \mathtt{a}^3), (\mathtt{a}^1, \mathtt{a}^5)\}$. Here, `a` is the identity of the variable and the index corresponds to the line number in the code.

When testing using the all-DU-path coverage criterion [17], DU-paths define test items that should be covered. Hence, should the software contain any unintended, and erroneous, DU-paths, these will be discovered by a full all-DU-path coverage testing. Coverage (i.e., the ratio between identified test items and exer-

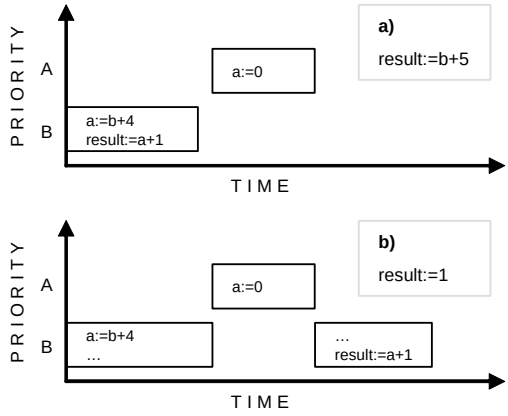


Figure 2. Shared variable communication (assuming that the conditional expression is true in task B).

cised test items) is the state-of-the-practice metric for test thoroughness. A 100% coverage describes a fully tested software with respect to a certain test criterion (e.g., branch, path, or DU-path coverage).

When moving from unit-level DU-based testing to system level testing of preemptive real-time systems, a whole new dimension of complexity is added - concurrent access to shared resources. Figure 2 shows an example execution of two concurrently executing tasks, A and B. B consists of the code from Figure 1. A has a higher priority than B, and contains a definition of variable `a`. In B there is a definition (`a:=b+4`) and two uses (`result:=a+1` and `result:=a*2`) of the same variable `a`. Hence, at task level, A only has a definition and no DU-paths. B has the following set of DU-paths: $\{(\mathtt{a}^{B1}, \mathtt{a}^{B3}), (\mathtt{a}^{B1}, \mathtt{a}^{B5})\}$, here indexed with the task identity and line number. Assume that, for correct intended behavior, B shall always complete the definition and the use in a sequence, i.e., A is not allowed to preempt B in between the definition and the use. Figure 2a illustrates the case without preemption and in-between definition of `a`. In Figure 2b, A preempts B and redefines the variable, thus corrupting the value. Using the task level DU-path sets for testing on system level will leave out scenarios as in Figure 2b where more paths evidently exist. In order to capture system level DU-paths, it is necessary to consider all scenarios (i.e., where the definition in A executes strictly before, strictly after, and where it preempts the definition and use in B). The resulting DU-paths are: $\{(\mathtt{a}^{B1}, \mathtt{a}^{B3}), (\mathtt{a}^{B1}, \mathtt{a}^{B5}), (\mathtt{a}^{A}, \mathtt{a}^{B3}), (\mathtt{a}^{A}, \mathtt{a}^{B5})\}$. In Figure 3, the scenarios from Figure 2 are revisited, but here, the focus is on the exact times when the accesses are executed. E.g., in Figure 3a definition `a:=b+4` is executed at def_1 and overwritten at rd_1. In Figure 3b,

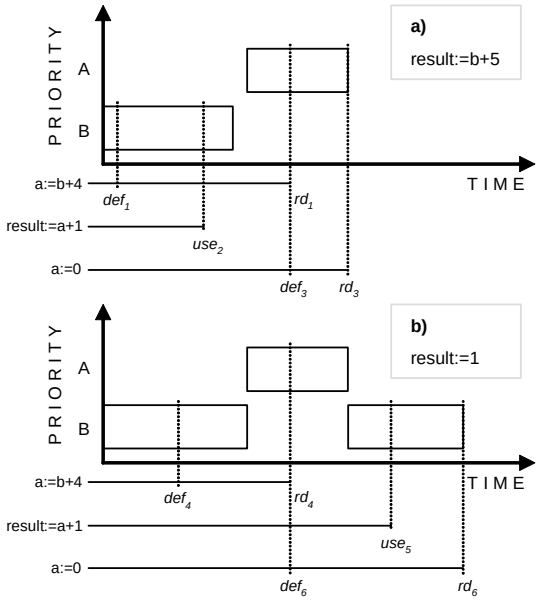


Figure 3. Shared variable access attributes.

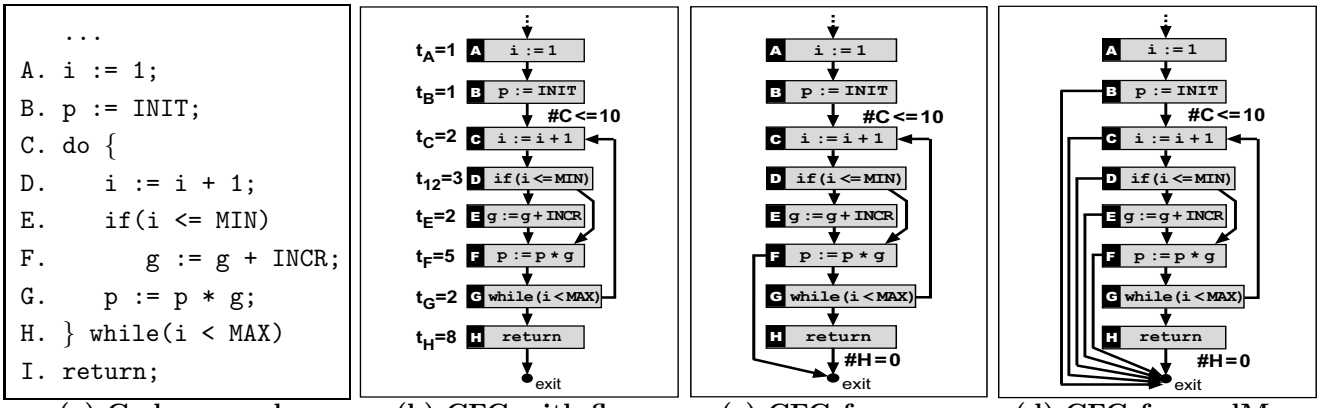
`a:=b+4` is executed at def_4 and overwritten at rd_4. Generally, for each access x, there is an interval with extremal values $x.min$ and $x.max$ within which x can be executed. Furthermore, for each definition d, there is a point in time $d.rdMax$, where d is safely overwritten. Hence, to derive feasible shared variable DU-paths on system level, we require (1) task-level information of when shared variables may be accessed by each task in the system, and (2) system-level information of how these tasks are scheduled and temporally interfere.

In this paper, we derive all system-level shared variable DU-paths by extending the method presented in [13]. The contributions of this article are:

- An extension to the SWEET WCET tool [6], able to derive task-level shared variable access times.
- The use of task-level shared variable access times for deriving system-level shared variable DU-paths.
- An experimental evaluation of the effectiveness of the approach.

We assume a uni-processor real-time system S. The operating system and application software (implemented as a set of tasks W_S) operates to control an external environment (e.g., a vehicular or industrial mechatronic control system). We assume strictly periodic tasks that follow the *single shot semantics* [2]. The tasks are scheduled using the *fixed priority scheduling* policy [1], and each task is assigned a unique priority.

In this paper, we represent a task as a 7-tuple, $\langle T, O, P, D, ET, \mathcal{D}, \mathcal{U} \rangle$, where T is the *periodicity* of the task. The task's release time for each period is calculated by adding the *offset* O to T. The scheduling



(a) Code example (b) CFG with flow and timing info (c) CFG for g use timing analysis (d) CFG for p rdMax timing analysis

Figure 4. Example of timing analysis for defs and uses.

mechanism determines which released task will execute based on the task's *priority*, P. The task's latest completion time is determined by its *deadline*, D. Further, a task encompass information regarding the best- and worst-case *execution time* of the task, ET, as well as two sets of shared data accesses, defined in terms of *definitions* ($\mathcal{D}$) and *uses* ($\mathcal{U}$) of the data. For each least common multiple of the tasks' period times (LCM), the system schedule performs a recurring pattern of task instance releases (*jobs*). In each LCM, each task can spawn one or more jobs. The release time R and deadline D of a job are calculated using the task T, O, and D properties respectively.

3 System-Level DU Analysis

In this section, we show how to derive information of when shared variables may be accessed by each task in the system (Section 3.1), and how to combine this information with the real-time schedule in order to derive all system-level DU-paths (Section 3.2).

3.1 Task-Level Analysis

The task-level analysis derives the temporal properties for each shared variable access (definition or use) in each task $w \in W_S$. Three properties (min, max, and $rdMax$) are derived for each definition, and two properties (min and max) are derived for each use. As min and max are analogous for definitions and uses, we will focus on the definition properties. Assuming a definition d that defines a variable x, the $d.min$ property describes the *shortest* possible time from the start of the task to the statement containing d. The $d.max$ property describes the *longest* possible time from the start of the task to the statement containing d. The $d.rdMax$ property describes the *longest* possible time for a path p, starting at task start s and ending at a statement e, such that d is on p, e contains a statement that redefines x, and no other redefinitions of x are made between d and e. Intuitively, this property describes the time (relative to the start of a task) where a definition d is safely overwritten.

The SWEET tool (SWEdish Execution time Tool) [5] is a research prototype WCET tool developed at Mälardalen University [10]. SWEET consists of three distinguished phases: a *flow analysis* where bounds on the number of times different entities in the code can be executed is derived, a *low-level analysis* where bounds of the execution times for instructions are derived, (taking into account the effects of pipelines and potentially instruction caches), and a final *calculation* phase where the flow and timing information is combined to yield a WCET estimate.

We have modified SWEET to, except the "normal" program WCET and BCET estimates, also produce estimates upon the above mentioned *min*, *max*, and *rdMax* values. Initially, we perform the flow- and low-level analysis of the program, but not the calculation. The result can be seen as a control-flow graph (CFG) containing both flow- and timing bounds and with two extra *start* and *exit* nodes. Figure 4(a) depicts an example code with two globals `g` and `p`. Figure 4(b) illustrates the CFG for the code. The flow analysis has derived a loop bound of 10, expressed as an upper bound on the number of times node `C` could be executed. Each node is also given a timing bound by the low-level analysis, valid each time the node is executed.

Secondly, we perform a *reaching definition* (RD) analysis for global variables [11]. The analysis derives, for each global variable, where in the program it may be used and defined as well as how far each definition may reach. Since pointers could be used to update globals, the RD takes the input of a pointer analysis.

We derive the different estimates using IPET calculation [5]. In IPET each node and/or edge in the CFG is given a time (t_{entity}), and a count variable (x_{entity}), the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. There are, e.g., constraints specifying that the *start* and *exit* nodes each must be taken exactly once, and constraints specifying that that each node must be entered the same number of times as it is exited. The estimate is normally derived using integer linear programming (ILP). The BCET is found by minimizing the same sum, subject to the same constraints.

Our analyses start from the above mentioned graph. Depending on what timing values to derive, we modify the graph by adding extra edges and flow contraints. E.g., the graph for deriving *min*, *max* for a use u is constructed by adding en extra edge from the node holding u to the *exit* node. Additionally, for all other edges going to the *exit* node we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through our newly created exit-edge, thereby deriving the best-case and worst-case estimates for u, instead of the "normal" BCET and WCET. Figure 4(c) shows the CFG for calculating *min* and *max* for the use of `g` in node `F`. For each global use and def derived in the RD analysis, we construct a corresponding graph. The resulting graphs are given as input to SWEET to derive the corresponding *min* and *max* values.

To derive *rdMax* for a def d we first use the RD analysis to derive the set of nodes which d may reach. From these nodes we add an extra edge to the exit node. Additionally, for all other exit-edges going from a node which d cannot reach, we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through *one* of the nodes d may reach. The *rdMax* value is derived by a WCET calculation upon the resulting graph. Figure 4(d) shows the graph for calculating the *rdMax* value of the `p := INIT` definition in node `B`.

3.2 System-Level Analysis

The algorithm for deriving system-level DU-paths is based on the algorithm deriving Execution Order Graphs (defined 1999 by Thane and Hansson [15] as directed reachability graphs of all possible execution orderings from a scheduled set of task instances during a periodically repeated FPS schedule). Basically, the EOG algorithm simulates the behaviour of a real-time FPS scheduler, considering all interleaving pattern alternatives caused by task execution time variations. As an example, Figure 2 displays two different execution orderings of the same system caused by execution time variations in task B. In our analysis, we modify the EOG algorithm such that it given our extended task model (with $\mathcal{D}$ and $\mathcal{U}$ properties) instead generates all possible DU-paths of the system. The algorithm simulates the behaviour of the system by exhaustively searching all task interleaving patterns, and acting upon shared variable accesses in the tasks at various times. Throughout the analysis, each access holds a certain state (*dead*, *active* or *live*). An *active* access has been executed, or can be executed at any time until it has become *live* or *dead*. A *live* access has safely been executed, and not been safely overwritten by another access. A *dead* access is neither *active* nor *live* (i.e., the access has safely not yet been executed, is safely overwritten, or has safely passed the time where it can affect the result of the analysis). The rules for making the transitions between these access states constitute the foundation of the system-level analysis:

Definition rules:

1. At $d.min$, d makes a transition from $dead \rightarrow active$.
2. At $d.max$, d makes a transition from $active \rightarrow live$.
3. At $d.rdMax$, d makes a transition from $live \rightarrow dead$.

Use rules:

1. At $u.min$, u makes a transition from $dead \rightarrow active$.
2. At $u.max$, u makes a transition from $active \rightarrow dead$.

DU-path rules:

1. At $d.min$, all DU-paths (d, u), such that $u.var = d.var$ **and** u is currently active, are derived.
2. At $u.min$, all DU-paths (d, u), such that $u.var = d.var$ **and** d is currently live **or** active, are derived.

These seven rules are implemented in the DUANALYSIS algorithm. This algorithm (Figure 5) is a slight variation[1] of the original EOG algorithm [16], built upon the manipulation of two data structures. Throughout the analysis, an abstract state of type **State** propagates through the execution of the system. For each execution of a job, the abstract state is changed according to the **Transition** created by executing the job. **State** represents the current abstract state of the execution, and contains information of currently live definitions, active definitions, active uses, and encountered DU-paths:

$$\textbf{State} \quad : \quad \{liveDefs, activeDefs, activeUses, duPaths\}$$

Transition represents a change of state incurred by the execution of a (partial) job. Thus, **Transition** contains new active definitions, killed live definitions, killed active definitions, killed active uses, and the

[1] Changes to the original algorithm are blackened in Figure 5.

```
DuAnalysis (state, transition, rdy, RI, SI)
{
      // When is the next job(s) released?
1.    t =NextRelease(SI)
2.    if rdy = ∅
3.       rdy = MakeReady(t, rdy)
4.       if rdy ≠ ∅
5.          DuAnalysis(state, transition, rdy, RI, (t, SI.r])
6.       else state = SwitchTask(transition, state, RI)
7.    else
         // Extract the highest priority job in rdy.
8.       J                  = Dispatch(rdy)
9.       [α, β)             = [max(J.R, RI.l), max(J.R, RI.l) + J.WCET)
10.      a'                 = α + J.BCET
11.      b'                 = β
12.      state              = SwitchTask(transition, state, RI)
13.      transition         = Execute(state, J, [α, β), RI)

         // Add all lower prio jobs released before J's termination,
         // or before a high priority job is preempting J.
14.      while((t < β) ∧ (Prio(t) < J.P))
15.            rdy    = MakeReady(t, rdy)
16.            t      = NextRelease((t, SI.r])

         // Does the next scheduled job preempt J?
17.      if ((t < β) ∧ (Prio(t) > J.P))
            // Can J complete prior to the release of the next job at t?
18.         if t > a'
19.            DuAnalysis(state, transition, rdy, [a', t), [t, SI.r])
20.            if rdy = ∅
21.               DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])
22.            else if t = a'
23.               DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])

            // Add all jobs that are released at time t.
24.         rdy = MakeReady(t, rdy)

            // Best and worst case execution time prior to preemption?
25.         J.BCET = max(J.BCET − (t − (max(J.R, RI.l)), 0)
26.         J.WCET = max(J.WCET − (t − (max(J.R, RI.r)), 0)
27.         Preempt(J, (t − (max(J.R, RI.l)), (t − (max(J.R, RI.r)))
28.         DuAnalysis(state, transition, rdy ∪ {J}, [t, t], (t, SI.r])

         // No preemption.
29.      else if t = ∞ // Have we come to the end of the analysis?
30.            DuAnalysis(state, transition, rdy, [a', b'), [∞, ∞]) // Yes

31.      else // More jobs to execute.
            // Is there a possibility for a high priority job to succeed
            // immediately, while low priority jobs are ready?
32.         if (rdy ≠ ∅ ∧ t = β)
33.            DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])
34.            if a' ≠ b' // And one branch for the low priority job.
                  // The regular succession of the next job
35.               DuAnalysis(state, transition, rdy, [a', b'), [t, SI.r))
}
```

Figure 5. The DuAnalysis algorithm.

WCET of the executed job:

$$\textbf{Transition} \;:\; \{newActiveDefs, killedLiveDefs, killedActiveDefs, killedActiveUses, wcet\}$$

Intuitively, the combination of a **State** a_1 and a **Transition** a_1' yields a new **State** a_2, representing the original state affected by the changes in a_1'. E.g., if a_1 contains a set of liveDefs $\{d_1, d_2, d_3\}$, and a_1' contains a set of killedLiveDefs $\{d_2\}$, then a_2's set of liveDefs will look as follows: $\{d_1, d_3\}$. In the algorithm, this process is formalized by the functions Execute and SwitchTask, where

$$\text{Execute} : \textbf{State} \times \textbf{Job} \times \textbf{Ivl} \times \textbf{Ivl} \rightarrow \textbf{Transition}$$

$$\text{SwitchTask} : \textbf{Transition} \times \textbf{State} \times \textbf{Ivl} \rightarrow \textbf{State}$$

In essence, the Execute function produces a change of abstract state (**Transition**) incurred on original abstract state by executing a certain job. The SwitchTask function produces a new abstract state (**State**), based on the original abstract state and the changes described by **Transition**. The implementation of these functions are directly based on the Definition, Use, and DU-path rules shown above. Two more structures (**Ivl** and **Job**) are used in the analysis. **Ivl** defines a time interval by its extremal values l and r. **Job** represents a task instance and contains definitions, uses, job priority, release time, BCET and WCET:

$$\textbf{Job} \;:\; \{\mathcal{D}, \mathcal{U}, P, R, BCET, WCET\}$$

Roughly, the DuAnalysis algorithm starts with an empty **State** at time 0 by scheduling the highest prioritized ready job j. Using the Execute function, j's **Transition** is derived. Next, if j is always finished before the next higher priority job is released, SwitchTask combines the **Transition** with the old **State** to a new **State**. The algorithm increments the time and schedules the next job. Else, if j is certainly preempted by a higher priority job, SwitchTask combines the **Transition** with the old **State** to a new **State** - but only regards the events that predate the preemption, stores the remainder of j, increments the time, and schedules the higher prioritized job. Else, if j *might* be preempted, the algorithm splits into two recursive branches, one of which considers the case with a preemption, and the other considers the case with no preemption. This behaviour is repeated until all jobs in the LCM are analysed. In order to derive the **Transition** created by executing a job j, the Execute function works through all shared variable accesses in j in a chronological order. Each access is treated according to its corresponding definition or use rule. SwitchTask creates a new **State** by adding the changes in j's **Transition** to the **State** prior to the execution of j. If j is not preempted, all changes in **Transition** are considered when creating the new **State**. Otherwise, only those changes prior to the preemption time are considered.

4 Evaluation

As an experimental evaluation of our method, we provide analysis results from five different multitasking real-time systems (S_1-S_5), each scheduled in three different ways (Cfg1-3). All systems comprise control-oriented code (e.g., calculation of planet orbits (S_1), a control system for a forklift able to solve the Towers Of Hanoi problem (S_2), etc.), and include intertask communication via shared variables. In Table 1, **Ts** and **GVar** refer to the number of tasks and global variables respectively. C_{DU} refers to the number of combinatorially feasible DU-paths (i.e., each definition d and use u of the same shared variable may naively

Sys	Ts	GVar	C_{DU}	Cfg1			Cfg2			Cfg3		
				F_{DU}	F_{DU} / C_{DU}	$rt(ms)$	F_{DU}	F_{DU} / C_{DU}	$rt(ms)$	F_{DU}	F_{DU} / C_{DU}	$rt(ms)$
S_1	4	18	216	155	71.6%	4282	147	68.1%	676	134	62.0%	19
S_2	4	27	183	N/A	N/A	N/A	176	96.2%	3750	163	89.1%	30
S_3	4	22	34	32	94.1%	217	32	94.1%	107	27	79.4%	4
S_4	3	7	44	34	77.3%	137	37	84.1%	24	24	54.5%	1
S_5	2	4	236	204	86.4%	422	196	83.1%	248	180	76.3%	12

Table 1. Evaluation Results.

form a DU-path (d, u)). F_{DU} refers to the number of DU-paths found feasible by the DuAnalysis algorithm, and rt refers to the analysis time (in milliseconds). Hence, 1 - (F_{DU} / C_{DU}) describes the percentage of DU-paths that safely do not have to be considered during testing. As for the configurations, Cfg3 completely separates all tasks in time and suffer no preemptions. In contrast, Cfg1 maximizes the number of task preemptions. Cfg2 is an in-between configuration of Cfg1 and Cfg3. Generally, with a less complex scheduling, more DU-paths are found infeasible (except for Cfg1 and Cfg2 of S_4). Note also that the system-level analysis of Cfg1 of S_2 proved too complex to execute. Since all other configurations finished within a few seconds, we will investigate this problem further.

5 Conclusion

For multi-tasking real-time systems, failures at system level caused by concurrently executing tasks cannot be revealed by tests at task level. In this paper, we have presented and evaluated a method that derives all possible DU-paths, enabling system-level testing of task inter-dependency failures. Our system model is restricted to a model suitable for small embedded real-time systems. In our future work we plan to show how our method can be used on a more relaxed system model, e.g., a system model based on transactions of tasks instead of strictly periodic tasks. Our method however requires a finite (periodically or non-periodically) repeated system behaviour. Further, semaphores and critical sections for shared variable access protection can be added to our method without major efforts as described in [14].

References

[1] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[2] T. Baker. Stack-based scheduling of real-time processes. In *Real-Time Systems Journal*, volume 3(1), pages 67–99, 1991.

[3] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. In *IEEE Software*, volume 8(2), pages 66–74, 1991.

[4] S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee. Testing of concurrent programs based on message sequence charts. In *Proceedings IEEE International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 72–82, Vol., Iss., 1999.

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

[6] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for Infeasible Path Calculation. In *Sixth International Workshop on Worst-Case Execution Time Analysis, (WCET'2006)*, Dresden, Germany, July 2006.

[7] M. Harrold and M. Sofia. Interprocedural Data Flow Testing. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 158–167, 1989.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[9] J. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. In *IEEE Transactions on Software Engineering*, volume 9(5), pages 347–354, May 1983.

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

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

[12] I. S. G. of Software: Engineering Terminology. IEEE Standards Collection, IEEE Std 610.12-1990. September 1990.

[13] A. Pettersson, D. Sundmark, H. Thane, and D. Nyström. Shared Data Analysis for Multi-Tasking Real-Time System Testing. In *Proceedings of Second Symposium of Industrial Embedded Systems*, July 2007.

[14] A. Pettersson and H. Thane. Testing of Multi-Tasking Real-Time Systems with Critical Sections. In *Proceedings of Ninth International Conference on Real-Time and Embedded Computing Systems amd Applications*, Tainan City, Taiwan, R.O.C, 18-20 February 2003.

[15] H. Thane and H. Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[16] H. Thane and H. Hansson. Testing Distributed Real-Time Systems. In *Journal of Microprocessors and Microsystems*, pages 463–478. Elsevier, 2001.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

Timing Analysis of Body Area Network Applications

Yun Liang Abhik Roychoudhury Tulika Mitra
Department of Computer Science, National University of Singapore
{liangyun,abhik,tulika}@comp.nus.edu.sg

Abstract

Body area network (BAN) applications have stringent timing requirements. The timing behavior of a BAN application is determined not only by the software complexity, inputs, and architecture, but also by the timing behavior of the peripherals. This paper presents systematic timing analysis of such applications, deployed for health-care monitoring of patients staying at home. This monitoring is used to achieve prompt notification of the hospital when a patient shows abnormal vital signs. Due to the safety-critical nature of these applications, worst-case execution time (WCET) analysis is extremely important.

1. Introduction

Embedded systems based on sensor networks are widely used in many contexts. Many applications running on sensor networks have real-time constraints. For example, Body Area Network (BAN) technologies are often applied in the health-care domain. Usually, the use of BAN in health care involves several low capacity sensor nodes on the human body and a powerful gateway device like a mobile phone or PDA. The nodes in this body area sensor network communicate through wireless connections and send data to the gateway. The gateway device can monitor the situation of the patient and inform the hospital if necessary in a timely fashion. Such sensor network based applications have stringent timing requirements.

Worst-Case Execution Time (WCET) is a required input to provide timing predictability in a system with timing constraints. Therefore, given an architecture and an application, estimating the WCET is very important for the system designer. In health-care domain, monitoring the patient's situation accurately and timely is so vital that the designer must perform WCET analysis for BAN applications.

In recent times, low-end sensor nodes such as the ones from Berkeley [3] have become popular and have been deployed in many applications. A typical example of a platform used by the sensor nodes is Tmote Sky [1]. Tmote Sky is a wireless sensor module for sensor network applications that require ultra low-power and high-reliability. A number of integrated peripherals including Timer, UART bus protocols, and DMA controller are provided by Tmote Sky.

The timing behavior of a BAN application on such sensor node architectures is determined by software complexity, program inputs, hardware and timing behavior of its peripherals. Hence, timing analysis for such sensor node architectures is non-trivial. This paper presents systematic timing analysis for BAN applications by integrating the timing behavior of each component on the platform and estimating the WCET of the application. First, the timing behavior of application code is analyzed through static timing analysis. This is done by extending *Chronos*, an existing timing analyzer for embedded software [4]. Next, the timing behavior of the peripheral devices are taken into account by analyzing the interrupt handler code and estimating the number of interrupts. This turns out to be extremely important in the context of WCET analysis of applications running on BAN.

The context of our work on BAN is a major programme on health-care monitoring being funded and carried out by Singapore's Agency of Science Technology and Research (A*STAR). The programme involves collaboration among many different projects — core technologies, middleware and applications. The aim is to develop and exploit embedded system technologies for health-care monitoring of patients staying at home (such that the hospital can be notified whenever any "unusual activity" in the patient's body is detected). Typical monitoring applications that we are studying include blood-pressure computation/detection (possibly for patients with cardio-vascular disease), fall detection (monitoring for elderly patients falling to the ground) and others.

The work described in this paper focuses on (a) the typical micro-architecture deployed in sensor nodes put on the patient's body, (b) modeling and timing analy-

sis of monitoring applications running on such sensor nodes, and (c) the significance of modeling the peripherals' timing behavior to accomplish a *full-system timing analysis* of such applications.

Related Work We are aware of one recent work on WCET analysis of applications running on sensor nodes [7]. The focus of this work is on the processor modeling and handling of nesC code. The applications considered are standard ones (sorting, sum, encryption, etc.) rather than from a specific domain. In terms of peripheral modeling, we know of one recent work on modeling a system controller [8] where the author develops a timing model for WCET analysis from the VHDL description of the system.

Organization of the paper The use of BAN for healthcare is outlined in Section 2. Section 3 presents a typical application based on BAN. Section 4 summarizes the key feature of Tmote sky architecture, which is a widely used sensor network platform. Static timing analysis based on Chronos is presented in Section 5. Finally, we present the results in Section 6 and provide concluding remarks in Section 7.

2. Body Area Network for Health Care

For some patients, their health conditions need to be monitored not only when they are at hospital, but also when they stay at home. Such daily medical report is very crucial for the doctors to diagnose some chronological diseases. Wearable systems can monitor the patients's condition by sensors placed on the body. Body Area Network technologies are used in such situations.

A Body Area Network comprises of some intelligent low-power devices including biomedical sensors and storage devices. The BAN works around the human body. The sensors are able to monitor and store important biomedical information and data. The BAN can also send information to the external world through a gateway, e.g, a PDA or mobile phone.

The overall architecture in which a BAN is deployed/used is shown in Figure 1, sensors on the human body communicate with both the gateway and other sensors.

3. BAN Applications

BAN applications are low-power applications. Hence, these applications keep the processor in low-power mode as much as possible and use *interrupts* to wake up the processor when data processing is required. Also, the peripherals are switched on only when needed in order to save energy. The

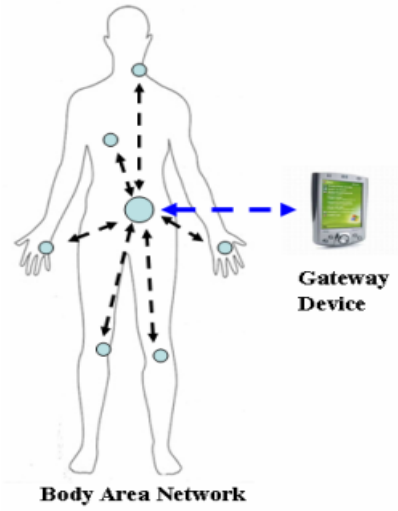


Figure 1: Body Area Network technologies for healthcare applications

high-level work-flow of a BAN application is as follows: *initialization, followed by a loop consisting of sampling and processing*, that is

initialization,
sampling, processing,
sampling, processing,
...

During sampling, the CPU could be turned off to save energy. When the sampled data is ready, it is processed by the application, and appropriate data is transmitted to remote devices. For analyzing such applications, we need to analyze the timing behavior of the main application and the interrupt handlers through static timing analysis.

The BAN application we analyze here continuously monitors the blood oxygen saturation level (SpO2) of a patient with *non-invasive* optical plethysmography also known as pulsoximeter. This measurement of the oxygen level and heart rate can be used to sound an alarm if they drop below a pre-determined level. This type of monitoring is specially useful in neonatal care and post-operative recovery. In a pulsoximeter, the estimation of blood oxygen saturation level (SpO2) is based on measuring the intensity of light that has been attenuated by body tissue. The amount of light absorbed by the body tissue depends on the oxygenation level of blood that is passing through it. Two different wavelengths of light are used — visible red wavelength and infrared wavelength.

The light intensity is sampled at a regular interval of 16ms. Every 16ms, first the red LED is turned on and the light passes through the finger of the patient to a photodiode. The intensity of light at the photodiode is

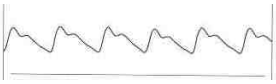


Figure 2: The waveform of sensor data.

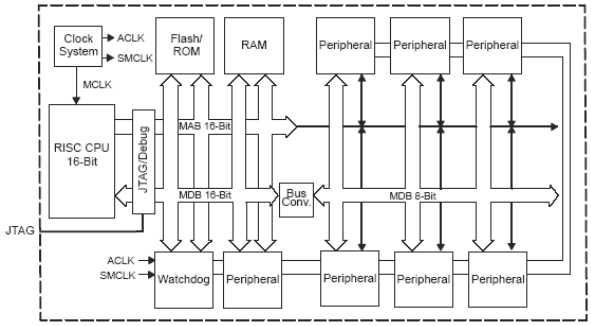


Figure 3: MSP430 Architecture ©Texas Instruments

sampled 16 times and the average value is taken. Next, the infrared LED is turned on and again the intensity of light at the photodiode is sampled 16 times to obtain an average. This process is repeated every 16ms. Figure 2 shows this average value of red light intensity attenuated by body tissue over time. The heart rate and SpO2 are estimated from these two waveforms corresponding to red and infrared light intensity. A window of a particular size is defined and the window slides on the waveforms. The application detects peaks in the waveforms inside the current window. The time interval between two peaks is used to measure both heart rate and SpO2 of the patient. These values are subsequently sent to some gateway device such as PDA that can detect abnormalities and send information to the hospital or health care provider.

4. Features of Tmote Sky Architecture

In this section, we outline the underlying micro-architecture of the sensor node platforms used by the BAN applications. Specifically, we describe the Tmote Sky platform [1], which employs the MSP430 processor from Texas Instruments. The MSP430 has a simple micro-architecture and a reduced instruction set with only 27 instructions. So, for WCET analysis, *the focus is not so much on the processor modeling.* Instead, modeling the timing effects of the peripherals turns out to be extremely important. We now describe the platform architecture in detail.

Tmote Sky is a mote platform designed for extremely low power, high data-rate, sensor network applications. The micro-controller present in Tmote Sky note is Texas Instruments's MSP430 F1611. The MSP430 incorporates a 16-bit RISC CPU, peripherals (12-bit ADC and DAC, Timer, USART, and a performance boosting DMA controller) and a flexible clock system. The architecture of MSP430 is shown in Figure 3. The current consumption of the micro-controller in low active and sleep mode is so small that an application can run for a very long time with only a single pair of AA batteries.

The features of Tmote sky platform with significant impact on the timing behavior of a BAN application are the following.

- **Flexible Clock System:** The platform includes a low-frequency auxiliary clock (ACLK) and a high-frequency master clock (MCLK). The peripherals can use different clocks.
- **Operating Modes:** MSP430 is designed for extremely low-power applications and features different operating modes distinguished by power, speed and current consumption. Operating mode can be selected by setting mode-control bits.
- **16-bit RISC CPU:** The number of CPU clock cycles required to execute an instruction depends on the instruction format and the addressing mode. All jumps instructions take two cycles to execute, regardless of whether the jump is taken or not. When computing execution cycles for interrupt handlers, the additional cycles due to interrupt overhead and reset should be taken into account.
- **Timer_A:** Timer_A is an asynchronous 16-bit timer counter with four operating modes. Clock source is configurable.
- **Timer_B:** Timer_B is identical to Timer_A with the exception that it can be programmed as 8, 10, 12, or 16 bit timer.
- **USART:** USART is used for asynchronous serial transmission and reception of characters to/from another device. The time to send/receive one character is based on the selected baud rate of the USART. Baud rate frequency is the same for both transmit and receive functions.
- **Hardware Multiplier:** The hardware multiplier is a peripheral and is not part of the 16-bit RISC CPU. The registers used by hardware multiplier are special peripheral registers.

The SpO2 application uses Timer_B, Timer_A and universal synchronous/asynchronous receive/transmit (USART). Timer_B is used to wake up the CPU and trigger Timer_A every 16ms. Timer_A is used to take 32 samples from the photodiode (16 for the red light and 16 for the infrared light) as discussed in Section 3. Timer_B and Timer_A use different clocks. USART is

used to send the heart rate and SpO2 measurements of the patient to some gateway device such as PDA.

5. Static Timing Analysis

The execution time of a program is determined by the program path taken during execution. If worst case input is known, then simulating the system with the worst case input will result in worst case execution time. However, determining the worst case input is very difficult when the application is non-trivial. Hence, static timing analysis is widely used technique to estimate worst case execution time. While many approaches have been proposed, we use Chronos [4, 5, 6] — an open-source timing analysis tool with detailed micro-architectural modeling developed by our research group. Given an architecture and an application, Chronos returns an upper bound on the execution time across all the inputs. WCET analysis in Chronos proceeds in two phases: *path-analysis* and *micro-architecture modeling*. The interested reader can get more details about (or even download) the Chronos toolkit from its website

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

An overview of the framework of Chronos is illustrated in Figure 4. During path analysis, Chronos constructs the control flow graph (CFG) of the application and generates functional constraints like loop bounds and flow constraints. In micro-architecture modeling, Chronos models complex micro-architectural features such as cache, pipeline, branch prediction and generate micro-architectural constraints. Following that, Chronos represents the execution time of the whole program through an Integer Linear Programming (ILP) formulation, and uses ILP/LP solver to find the maximum execution time.

We modify the Chronos toolkit in order to model the micro-controller MSP430. Chronos is targeted towards SimpleScalar PISA instruction-set architecture (ISA). As MSP430 has a different ISA, the control flow graph construction is modified. The other components in path analysis such as flow constraints generation and loop bound detection remain the same. While Chronos models cache, pipeline, branch prediction, these modeling are omitted here as MSP430 processor does not support these features. The execution cycles corresponding to an instruction is obtained by a simple table look-up with the corresponding instruction format and addressing mode. Then, the execution time of a basic block is simply the sum of the execution time of the instructions within the basic block.

The main contribution of this work is that we extend Chronos to model and analyze the timing behavior of the peripherals. For SpO2 application, the peripherals include Timer_B, Timer_A and USART. The next section describes the modeling of the peripherals in detail.

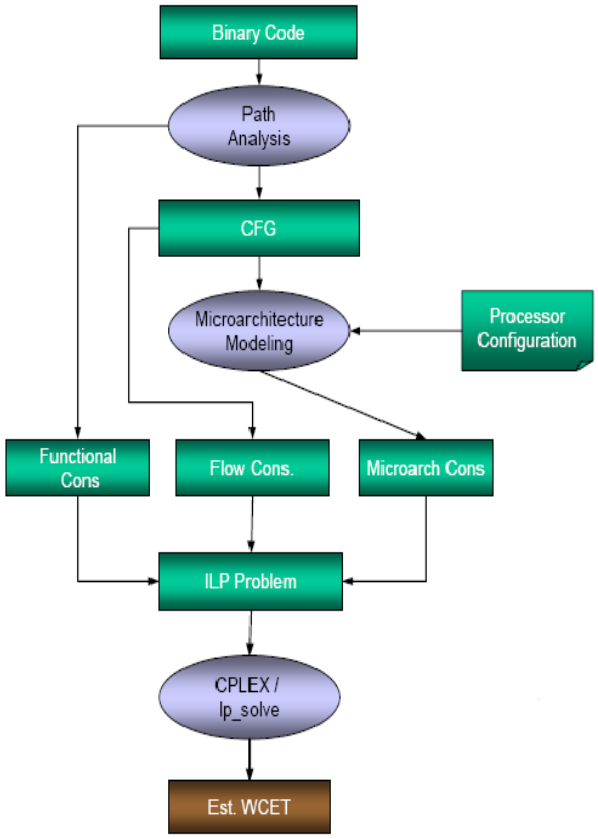


Figure 4: Workflow of Chronos Timing Analysis Tool

6. System-Level Modeling for BAN Application

The interesting aspect of any BAN application is that the sensor nodes are sampled at regular intervals through the use of timers. In other words, the number of interrupts can be estimated accurately. Therefore, the timing behavior of the main application and the interrupt handlers for Timer_A and Timer_B can be analyzed separately. We use Chronos to perform static timing analysis of the interrupt handlers and the main application individually. However, we should ensure that the timing overheads due to interrupt acceptance and return are taken into account. Finally, summing up the WCET of all the sub components yields the WCET of the whole system. Here, we define WCET as the total time required to acquire and process the samples every 16ms. The timing analysis of the interrupt handlers and the main application for SpO2 are illustrated

Component	Clock System	WCET cycles	WCET	WCET Percentage	ACET
Processing	High Speed (8 MHz)	64352	8.044 ms	61.73%	0.586 ms
Timer_B	Low Frequency (32 KHz)	38	4.75 μs	0%	4.75 μs
Timer_A	High Speed (8 MHz)	35724	4.4655 ms	34.27%	3.766 ms
USART	Asynchronous Transmission		0.52ms	4%	0.52 ms
Total			13.03ms	100%	4.872 ms

Table 1: Results of Timing Analysis of SpO2 application with interrupt

in the following.

Timer_B: Timer_B uses the low frequency auxiliary clock (ACLK) running at 32 KHz. The counter for this timer is set such that it generates an interrupt every 16ms. The functionality of Timer_B interrupt handler is to simply enable Timer_A to sample data. The WCET of Timer_B interrupt handler routine is shown in Table 1.

Timer_A: Timer_A is enabled by Timer_B. As mentioned earlier, each time Timer_A takes 32 samples from the photodiode (16 at red light wavelength and 16 at infrared wavelength). Finally, two average values are computed from these samples. The interrupt handler is invoked for each sample. However, only 2 out of these 32 invocations lead to average value computation, which is more time consuming. So, the call context is taken into account when analyzing the worst case execution time of the Timer_A interrupt handler. The WCET of Timer_A handler routine for one round (i.e., 32 samples) is shown in Table 1. When the average values have been computed, the CPU exits the low power mode and returns to active mode for data processing.

Processing: In the main application, the average values obtained from Timer_A are filtered and stored into a window. Then, the application tries to detect a peak in the middle of this window. If a new peak is detected, heart rate and SpO2 are calculated using the peak value. Finally, USART is called to send some feedback to the gateway device. From the description above, it is obvious that the worst case behavior happens when a peak is detected. The WCET of the main application (Processing) is shown in Table 1.

USART: The USART takes 1/115200 second to send 1 bit and SpO2 needs to transmit 6 bytes per round. In the USART, for every byte of data, two more bits (start bit and stop bit) are added. Therefore, a total of 60 bits are transmitted per round. The time spent in the USART is shown in Table 1. In SpO2 application, the USART transmission takes place via polling; so the time spent in the USART is the blocking time due to transmission.

From the Ratio column in Table 1, we find that the time consumed in peripherals is 38.27% for one round of acquisition and processing. *Clearly, if the timing effects of the peripherals are not modeled, the WCET of the whole system will be an under-estimation.*

The average case execution time (ACET) is shown in Table 1 too. In terms of ACET, the time for one round of data acquisition and processing is close to 4.872 ms. Clearly, the WCET value is essential here to claim that timing constraint is satisfied.

7. Conclusion

In this work, we present a systematic timing analysis framework for BAN applications. BAN applications are safety-critical and have stringent timing requirements. We analyze the timing behavior of application code and interrupt handlers for peripherals separately. Finally, the WCET of the interrupt handlers are multiplied by the number of interrupts during one round of processing and summed up with the WCET of the main data processing part to estimate the WCET of the entire system.

TinyOS [3] is an operating system designed for wireless embedded sensor networks and has been used widely for sensor nodes. NesC [2] is the corresponding language for programming sensor network applications in TinyOS. In the future, we plan to adapt our framework to provide WCET analysis for NesC code along with modeling of TinyOS.

Acknowledgments

This work was supported by NUS project R252-000-171-112 and A*Star SERC EHS-II project R-252-000-258-305.

References

[1] Tmote sky platform. http://www.moteiv.com/products/.

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach

to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[4] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007.

[5] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time System.*, 29(1):27–58, 2005.

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

[7] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 405–414, Washington, DC, USA, 2005. IEEE Computer Society.

[8] S. Thesing. Modeling a system controller for Timing Analysis. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 292–300, New York, NY, USA, 2006. ACM Press.

Data-Flow Based Detection of Loop Bounds

Christoph Cullmann and Florian Martin
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
{cullmann,florian}@absint.com, http://www.absint.com

Abstract

*To calculate the WCET of a program, safe upper bounds on the number of loop iterations for all loops in the program are needed. As the manual annotation of all loops with such bounds is difficult and time consuming, the WCET analyzer **aiT** originally developed by Saarland University and AbsInt GmbH uses static analysis to determine the needed bounds as far as possible.*

*This paper describes a novel data-flow based analysis for **aiT** to calculate the needed loop bounds on the assembler level. The new method is compared with a pattern based loop analysis already in use by this tool.*

1. Introduction

To calculate the WCET for a program, safe upper bounds for the iterations of all included loops must be known. To get a precise WCET estimation, lower bounds should be known, too.

As programs tend to contain many loops with bounds depending on the call sites of the surrounding routine, relying on user annotations for loop bounds would cause too much work for the user. Beside that, there is also the inherent danger that user-annotated bounds could contain errors, as they need to be kept up to date while the application code is changing. Therefore **aiT** aims at deriving safe loop bounds automatically by using a static analysis.

Until now, a pattern-based approach for loop bound detection is used. This method needs adjustments for all supported compilers and in some cases even different optimization levels. While experience has shown that this works well for many simple loops, no bounds are detectable for more complex loops with multiple modifications of the loop counter inside one iteration.

To overcome these restrictions, we introduced a new method for loop bound detection that uses an interprocedural data-flow analysis to derive loop invariants from the semantics of the instructions. This new analysis does not depend on the used compiler or optimization level but only on the semantics of the instruction set for the target machine. It is able to handle loops with multiple exits and multiple modifications of the loop counter per iteration including modifications in procedures called from the loop. Additional, it detects and handles overflows of size limited datatypes.

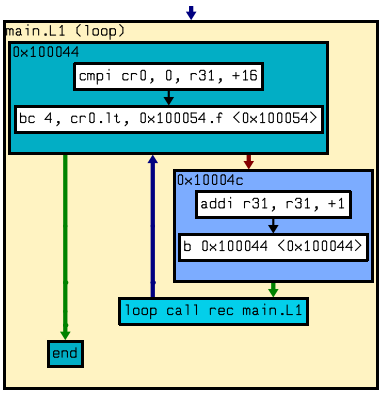
In this section, we describe the techniques behind the old and new loop analyses, compare their results, and provide insight on how the new analysis will be used in **aiT**. First we start in Section 2 with introducing the common basis of both analyses. In Section 3 two small examples for loops are shown that will be used later as running examples to illustrate the application of both analyses. Section 4 will cover the pattern-based approach. Then we introduce the new data-flow based approach in Section 5 and compare both analyses in Section 6. Finally we show how the new analysis is integrated into the WCET Analyzer **aiT** in Section 7.

2. Common Basis for Both Analyses

As both loop analyses have been developed to be used as part of the WCET Analyzer **aiT**, they are using the **aiT** framework presented in [3]. In particular, they operate on a control flow graph which is reconstructed from the machine executable (see [10]) and in which all loops have been transformed to tail-recursive routines by a loop transformation (described in [7]). The next section will show two example loop routines, which are used in the subsequent description of both analyses. A loop iteration equals one execution of the loop routine.

While the **aiT** framwork and the presented loop analyses work on the compiled executables, there are other approachs that work on the level of the programming language. For example in [6] and [5] a framework is described that works on the C sources of a program to calculate a WCET and the therefore needed loop bounds.

To avoid code duplication, the analyses use the existing value analyzer of the framework to query the addresses of




```
while (r31 < 16) // 0x100044
{
  r31 = r31 + 1; // 0x10004c
}
```

Figure 1. A loop with one loop test and single increment

memory accesses and to obtain knowledge about the contents of accessed registers and memory cells. As the value analysis produces integer intervals as approximations for addresses and memory contents, both loop analyses use intervals for their calculations, too. Beside this, the loop analyses query the value analysis for infeasible control-flow edges, i.e. edges that are not taken in any run of the program. This information is used in both analyses to exclude unreachable loops from loop bound detection. For more details about the value analysis please refer to [9]. The value analysis information allows separate analysis of the loops for each calling context and monitoring the loop counter even if it is a global variable, a function parameter or modified over a pointer, which is important as shown in [8].

The analyses take into account that programs often contain nested loops for which the iteration bounds of the inner loops depend on the iteration bounds of outer loops. Therefore both analyses sort the loops by their nesting depth and analyze them from the outside to the inside. After handling one nesting depth, value analysis is restarted with the new derived loop bounds as input to get more precise information while looking for the bounds of the inner loops.

As value analysis gets more precise if it also knows the lower bound of a loop, both analyses output not only the safe upper bounds needed to calculate any WCET, but intervals that are guaranteed to contain all possibilities for the number of loop iterations.

3. Running Examples

To illustrate the working of the two loop analyses, two simple loops found in programs for the *PowerPC* architecture are chosen as examples. Figures 1 and 2 show the corresponding loop routines.

Both loops use machine register 31 as their loop counter. We assume for the upcoming calculations and analyses that this register contains the value zero before the first loop iteration.

The loop in Figure 1 is a simple loop incrementing its loop counter in each iteration by exactly one. The loop is first entered with counter value 0, then with value 1, etc. until it reaches 16. When it is entered with counter value 16, the test `r31 < 16` fails for the first time so that there are no further loop iterations. Therefore, there are exactly 17 loop iterations. The loop analysis should thus return the interval $[17, 17]$ (the most precise answer) or any larger interval containing 17 (correct, but imprecise).

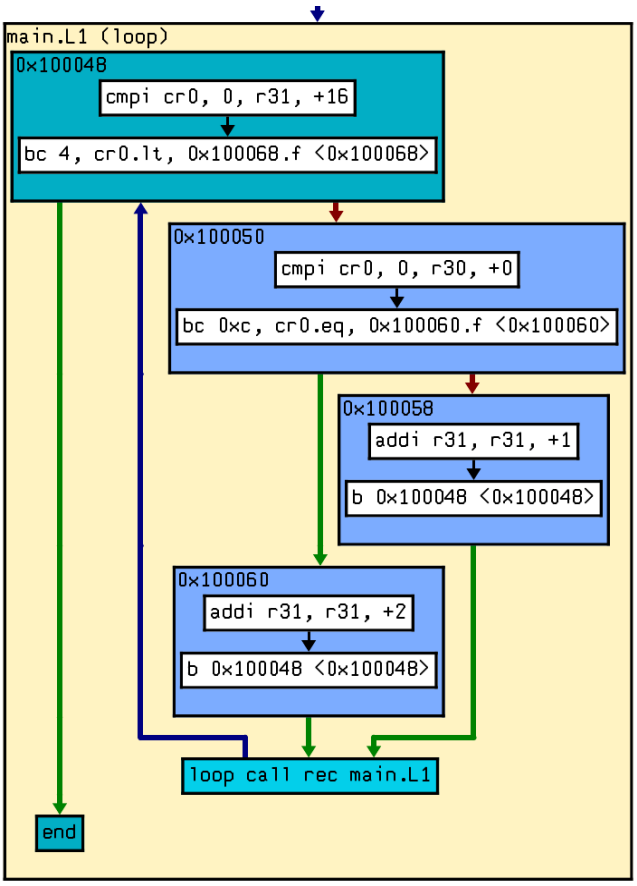
The loop in Figure 2 is similar, but a counter increment of one or two is possible, as the control flow forks into two branches inside the loop routine. The safe upper bound is still 17 as in the first example, but the lower bound is now only 9. The result of the loop analysis should thus be $[9, 17]$ or any larger interval.

4. The Pattern-Based Approach

The current loop analysis in **aiT** uses patterns to detect the loop bounds for common loop variants. These patterns are handcrafted for the supported compilers and their different optimization levels. Some intraprocedural analyses are used to handle the matching, like intraprocedural slicing and dominator/postdominator analysis.

A typical loop pattern to detect loops generated by C compilers from `for`-loops consists of the following conditions:

- The loop is only left by one conditional branch;
- the same compare of a register with a constant sets the condition for this branch in each iteration;
- the register that is compared is incremented by a constant value at the same instruction in each iteration;
- the start value of the register is known by the value analysis.




```
while (r31 < 16) // 0x100048
{
  if (r30 == 0) // 0x100050
  {
    r31 = r31 + 1; // 0x100058
  }
  else
  {
    r31 = r31 + 2; // 0x100060
  }
}
```

Figure 2. A loop with one loop test and two different increments

To match even such a simple pattern, multiple internal subanalyses must be performed. For this example pattern, the following steps would be needed:

- Check for a conditional branch instruction that dominates and postdominates the recursive call of the loop routine;
- slice backwards from the branch inside the loop routine to find the compare instruction modifying the condition flag evaluated by the branch instruction;
- test whether it is a compare of a register with a constant;
- slice backwards from the compare instruction to find all instructions modifying the registers/memory cells used in the compare instruction;
- test whether only one instruction is found in the last step and whether it is a constant addition/subtraction;
- test whether this one instruction dominates and postdominates the compare instruction;
- query the value analysis for the start value of the used register;
- calculate the bounds by using the now known start/end value and increment.

If we apply this pattern to our example loop of Figure 1, we get a match, as this loop is left only by a conditional branch after the compare of the loop counter with some constant and the loop counter is incremented in each round by one. The resulting bound would be $[17, 17]$, which is in this case the optimal solution.

The slightly more complex loop of Figure 2 is not matched by this pattern, as the loop counter is not incremented in each iteration by the same instruction, but by two different `addi` instructions in two different control-flow branches. Therefore no loop bound can be determined and thus no WCET is obtained.

Given how many steps are already needed for this simple pattern and that all this needs to be done by handwritten code, it is clear that bigger patterns to handle more complex loops, like the one shown above, are time consuming to implement correctly and to maintain. This illustrates the need

for a new kind of loop analysis, which will be presented in the next section.

5. Improved Loop Analysis Based on Data-Flow Analysis

To enhance the loop bound detection for more complex loops and to avoid the dependencies on compiler versions and optimization levels, a new loop bound analysis based on data-flow analysis was designed. The following provides a brief introduction to this new method. More information can be found in [2].

A run of the new analysis consists of the following phases:

1. Classification of all loops;
2. Detection of possible loop counters;
3. Data-flow analysis to derive the invariants;
4. Analysis of the loop tests to calculate the loop bounds.

5.1. Loop classification

In the first phase, loops are classified using information obtained from value analysis. Loops that can never be reached are excluded from further analysis and get the safe bound $[0, 0]$ as the corresponding loop routines are never called. For the remaining loops, the algorithm checks whether value analysis already knows after how many recursive calls their loop routine cannot be called again. If this number is known, it can be taken as a safe upper bound for the loop, even if the further stages fail to produce results.

5.2. Search for possible loop counters

For the loops that still need to be analyzed, a simple intraprocedural analysis is run to search all registers and memory cells accessed inside the loop routine. Then it is checked whether value analysis knows their start value, i.e. their value before the first call of the loop routine. The registers and memory cells with known start value are considered as potential loop counters. They are further examined by a data-flow analysis to derive loop invariants (see below). Loops without any detected loop counter must remain unbounded.

Our first example loop (Figure 1) only accesses register 31. For our second example (Figure 2), the intraprocedural analysis would find registers 30 and 31. Assuming that value analysis only knows the start value of register 31, this register would be the only potential loop counter in both loops.

5.3. Invariant analysis

This data-flow analysis is the core of the improved loop analysis. For each potential loop counter detected in the previous phase, it calculates for each program point of the loop routine a set of expressions, called invariants, that indicate how the counter is modified from the entry of the loop routine to this point in each iteration.

The analysis uses a special language for the expressions, *IVALA*. Variables in *IVALA* expressions describe registers or memory cells, including information about the register number or memory address and the data size in bytes. The loop counter in our examples would be expressed in *IVALA* as $(register, 31, 4)$, as it is register 31, which is 4 byte wide.

The language allows to express assignment between variables, assignment of a constant integer interval to a variable, and modification of a variable by adding a constant integer interval. This seems to be very restrictive, as other modifications like non-constant addition or any kind of multiplication are not supported, but the evaluation in the next section will show that it is sufficient to detect most loop bounds in a program, as the most common loops are counting loops. Besides, this restriction serves to keep the complexities of invariant analysis and of the subsequent bound calculation within reasonable bounds.

For the loop routine of our first example shown in Figure 1 the analysis would e.g. calculate the following expression set for the ingoing edge of the recursive call of the loop routine:

$$\{(register, 31, 4) \; = \; (register, 31, 4)^{\circ} + [1, 1]\}$$

where $(register, 31, 4)^{\circ}$ is a placeholder for the value of $(register, 31, 4)$ at the beginning of the loop iteration. The expression indicates that register 31 is incremented by exactly one in each iteration. For the example in Figure 2 the analysis would calculate:

$$\{(register, 31, 4) \; = \; (register, 31, 4)^{\circ} + [1, 2]\}$$

This provides the information that the register is incremented by one or two.

5.4. Evaluation of the loop tests and bound calculation

In this phase, for each loop all existing loop tests will be evaluated. A loop test is a basic block with a conditional branch leaving the loop routine. For each test a bound will be calculated. All these bounds are then combined to one bound for the whole loop. The following steps are needed to calculate the bound for a loop test:

- The branch type is determined;

- the compare instruction evaluating the condition used by the branch is searched;
- the variables used in the compare instruction are detected;
- the flow-analysis results are used to get expressions for the found variables;
- an equation system is built and solved to get the concrete loop bound.

A detailed description of this process can be found in [2].

For our first example (Figure 1), this process would look as follows:

- Inspection of the branch in basic block `0x100044` yields that the loop is left on greater-equal.
- A search for the corresponding compare instruction finds the first instruction in the block.
- As variable $(register, 31, 4)$ and the constant integer 16 are used, the exit expression is $(register, 31, 4) \geq 16$.
- The flow-analysis will yield that $(register, 31, 4)$ is incremented by one in each iteration.
- The solver will compute the concrete bound $[17, 17]$, which is the optimal solution.

The handling of the second example is analogous, except that the flow-analysis delivers an increment of $[1, 2]$ and therefore the solver would calculate the bound $[9, 17]$.

Both examples show comparisons with integer constants as loop test but comparisons of two variables are supported, too, as long as the value analysis is able to detect a constant interval for one of them.

6. Practical Evaluation

While the new analysis is more generic by design, we still need to demonstrate that it is applicable to real-world programs. Therefore an extensive evaluation with both code from a compiler benchmarks suite and with real software from the embedded-system world was performed in [2].

The results show that the new analysis method works for most loops equally well or better than the pattern-based method. Only in some corner cases, the old analysis takes the lead, as it has special patterns for them.

The runtime costs of both analyses are comparable: the new analysis is slower than the pattern-based approach only by a constant factor of at most three for some tests. Table 2 shows measured runtimes of both analyses for four different tasks out of industrial real-time software for a *PowerPC MPC755*. The runtimes were measured on a 3.2 GHz Pentium 4 with 2 GB RAM running *Linux*.

test	optimal	old analysis	new analysis
do_char_001	$[1, \infty]$	$[1, \infty]$	$[1, \infty]$
do_char_008	$[16]$	$[16]$	$[16]$
do_char_009	$[16]$	$[16]$	$[1, \infty]$
do_char_010	$[1, 16]$	$[1, 16]$	$[1, \infty]$
for_char_001	$[17]$	$[1, \infty]$	**[17]**
for_char_017	$[17]$	$[17]$	$[17]$
for_char_049	$[1]$	$[1, 17]$	$[1, 17]$
for_char_058	$[17]$	$[1, \infty]$	**[17]**
for_char_061	$[9]$	$[1, \infty]$	**[9]**
for_char_062	$[17]$	$[1, \infty]$	**[17]**
for_int_001	$[17]$	$[1, \infty]$	**[17]**
for_int_017	$[17]$	$[17]$	$[17]$
for_int_049	$[1]$	$[1, 17]$	$[1, 17]$
for_int_058	$[17]$	$[1, \infty]$	**[17]**
for_int_061	$[9]$	$[1, \infty]$	**[9]**
for_int_062	$[17]$	$[1, \infty]$	**[17]**

Table 1. Single loop synthetic tests, *DiabData*

test	old analysis	new analysis
mpc755_1	43.54	63.75
mpc755_2	3.82	9.25
mpc755_3	0.53	0.77
mpc755_4	0.47	0.69

Table 2. Runtimes of analyses in seconds

To show that the new analysis is compiler-independent, Tables 1 and 3 present the results of both analyses for code generated by the *DiabData* ([11]) and *GNU* C compiler ([4]), respectively. While both analyses work reasonably well for the DiabData compiler, only the data-flow based analysis works for the GNU C compiler without adjustments. To obtain comparable results, the pattern-based analysis would require additional effort to develop loop patterns adapted to the code generated by the GNU C compiler.

7. Summary and Outlook

As the evaluation has shown, both analyses have some benefits in their own areas. While the pattern-based analysis can keep the lead for special cornercases where handcrafted patterns can play out their strength, the data-flow based analysis works best for typical loops occurring in standard programs. This flexibility of the new analysis is reached by it's expressions, which are powerful enough to handle loops with multiple exits, multiple/conditional changes of the loop counter and overflows of the used datatypes.

As **aiT** is aimed to provide the best loop bound detection possible, both analyses will be used in combination. First

test	optimal	old analysis	new analysis
do_char_001	$[1, \infty]$	$[1, \infty]$	$[1, \infty]$
do_char_008	$[16]$	$[1, \infty]$	$\mathbf{[16]}$
do_char_009	$[16]$	$[1, \infty]$	$\mathbf{[16]}$
do_char_010	$[1, 16]$	$[1, \infty]$	$\mathbf{[1, 16]}$
for_char_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_017	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_049	$[1]$	$[1, \infty]$	$\mathbf{[1, 17]}$
for_char_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_char_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_017	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_049	$[1]$	$[1, \infty]$	$\mathbf{[1, 17]}$
for_int_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_int_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$

Table 3. Single loop synthetic tests, *GNU*

the fast pattern-based analysis is applied, and only for the loops it is not able to handle, the more generic new analysis is run. This avoids any slow down for the analysis of programs for which the old analysis already detected all bounds, and enables the calculation of the WCET for programs with more complex loops.

This combined strategy is already in use for the *PowerPC* and *M32* architectures, with plans to extend it to the *VAMP* architecture (described in [1]) in the near future.

References

[1] S. Beyer. *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, 2005.

[2] C. Cullmann. Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Diploma Thesis, Universität d. Saarlandes, 2006.

[3] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[4] GNU Project. *GCC Version 3.3*, 2006.

[5] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A tool for automatic flow analysis of c-programs for wcet calculation. In B. Werner, editor, *In Eight IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 106 – 112, Guadalajara, Mexico, January 2003. IEEE.

[6] J. Gustafsson, B. Lisper, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c programs. In G. Bernat, editor, *WCET 2002 Workshop*, Vienna, June 2002.

[7] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the International Conference on Compiler Construction (CC'98)*. Springer-Verlag, 1998.

[8] C. Sandberg. Inspection of industrial code for syntactical loop analysis. In *WCET 2004 Workshop*, Catania, July 2004.

[9] M. Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität d. Saarlandes, 1997.

[10] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000.

[11] Windriver. *DiabData C Compiler Version 4.4*, 2006.

Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis[†]

Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{andreas.ermedahl,christer.sandberg,jan.gustafsson,stefan.bygde,bjorn.lisper}@mdh.se

Abstract

Static Worst-Case Execution Time (WCET) analysis is a technique to derive upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component for static derivation of precise WCET estimates is upper bounds on the number of times different loops can be iterated.

In this paper we present an approach for deriving upper loop bounds based on a combination of standard program analysis techniques. The idea is to bound the number of different states in the loop which can influence the exit conditions. Given that the loop terminates, this number provides an upper loop bound.

An algorithm based on the approach has been implemented in our WCET analysis tool SWEET. We evaluate the algorithm on a number of standard WCET benchmarks, giving evidence that it is capable to derive valid bounds for many types of loops.

1 Introduction

The WCET is an important parameter when verifying real-time properties. A *static WCET analysis* finds an upper bound to the WCET of a program by analysing the statical properties of the hardware and software involved. Given that the methods used are correct and safe, the analysis will derive a timing estimate that is *safe*, i.e., a value $\geq$ WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as the execution time of individual instructions, as well as the program's *possible execution flows*, to bind the number of times the instructions can be executed, needs to be derived. The latter includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc.

The goal of *flow analysis* is to calculate such *flow information* as automatically as possible. Flow analysis research has mostly focused on *loop bound* analysis, since upper bounds on the number of loop iterations must be known in order to derive WCET estimates. Recent industrial WCET case studies [9] have shown that it is important to develop good support for flow analysis, in particular loop bound analysis, in order to reduce the need for manual annotations.

This article presents an approach how to calculate upper loop bounds statically. The approach builds on the observation that terminating loops always must reach a new state for each new iteration. Thus, if we can somehow bound the number of states which are possible to reach during any execution of the loop, then that number provides an upper bound to the number of loop iterations *provided that the loop terminates*. Since in general many states may be equivalent w.r.t. program flow, it suffices to count the number of equivalence classes of states. An upper bound to the number of possible equivalence classes is the number of possible combinations of values for variables affecting the exit conditions of the loop.

Based on this observation, we perform a loop bound analysis using a combination of standard program analysis techniques:

1. By *program slicing* we derive a set of variables and statements that must be considered when deriving a loop bound for a given loop. Only this code is analysed.
2. By *abstract interpretation* (AI) we derive, for each program point and variable, an upper approximation of the possible set of values held by the variable in that program point. This information can be used to limit the possible number of (equivalence classes of) states in loops.

[†] This work has been supported by the KK-foundation through grant 2005/0271. It has also been funded in part by the ARTIST2 Network of Excellence (www.artist-embedded.org).

3. A variable can sometimes have many possible values in a loop, altough its value will always remain the same for each execution of the loop. By *invariant analysis* we identify variables not having their values changed in loops. These variables are removed from the corresponding loop bound calculations.

All the analyses above terminate. Thus, the suggested loop bound analysis also terminates. Moreover, since AI is input dependent, (bounds on input values can be specified), the analysis is also input dependent.

Since our approach assumes that analysed loops terminate, separate proofs of termination will have to be provided. In Section 7 we discuss this problem.

The reminder of this article is organized as follows: Section 2 presents related work. Section 3 presents an illustrating example. Section 4 gives more details of the approach, including some implementation details. Section 5 presents our WCET tool. Section 6 presents some evaluations of the approach. Finally, Section 7 gives our conclusions and ideas for future work.

2 Related work

Upper bounds on the number of loop iterations are needed in order to derive a finite WCET estimate at all. Similarly, recursion depth must also be bounded. Due to the halting problem, no automatic method for loop bounds analysis can give an exact answer for all loops. Thus, WCET analysis tools provide means to give loop iteration bounds manually [5, 6, 17]. However, this is often laborious, and a source of possible errors.

Although necessarily incomplete, an automatic loop bounds analysis can still be useful to reduce the manual work by bounding most of the commonly occurring loops. A common approach is to identify loop counters, and then determine (or bound) their start values, increment (decrement), and highest (or lowest) possible value. From this information, an upper bound for the iteration count can be obtained. Whalley et al. [12] use data flow analysis and specialized algorithms to calculate loop bounds for both single and some special types of nested, triangular loops. This approach is quite syntactical and will fail for loops which do not fit the patterns. The loop-bound analysis of the Bound-T tool [17] estimates range and increment for loop counters using Presburger arithmetics, and the latest loop bound analysis of the aiT tool [4] decides start values by an interval-based AI and the possible increments by a data flow analysis. These methods have in common that they only work for well-structured loops with a proper nesting, and where loop counters are updated using addition or subtraction only. In contrast, our analysis is based entirely on abstract interpretation,

```
1.  int foo(int INPUT) {    // INPUT = [10..20]
2.      int OUTPUT = 0;
3.      int i = 1;
4.      while(i <= INPUT) {   // p
5.          OUTPUT += 2;
6.          i++;
7.      }
8.      return OUTPUT;
9.  }
```

Figure 1. Illustrative code example

which makes is less sensitive to the kind of operations applied to loop counters. It works also for unstructured loops without proper nesting.

We have previously presented *Abstract Execution* (AE), a form of symbolic execution based on an AI framework. AE is able to derive loop bounds and infeasible path information for many type of programs [10, 11] but has a potential bad worst-case complexity, and no guaranteed termination. Thus, the approach presented in this article complements the AE by being less general, since it cannot bound all type of loops, but with guaranteed termination.

3 An illustrative example

As an illustrative example, consider `foo` in Figure 1. The `INPUT` variable is given the value limit [10..20].

A slicing w.r.t. the exit condition of the loop discovers that `OUTPUT` does not affect the outcome of the condition, and could, together with statements `2`, `5` and `8`, be sliced away. `i` and `INPUT` are both used in the loop exit condition and must therefore be kept.

An AI using the interval domain together with widening and narrowing (see Section 4.2) will find that at program point `p`, the possible values of `i` lie in the range [1..20], and those of `INPUT` in [10..20]. Thus, a safe bound on the number of times the loop body can be executed, given that the loop terminates, is given by $\texttt{size(i,p)} * \texttt{size(INPUT,p)} = \texttt{size}([1..20]) * \texttt{size}([10..20]) = (20-1+1)*(20-10+1) = 20*11 = 220$. Here $\texttt{size}(v,p)$ is the number of possible values of variable v at a program point p as given by the AI.

We improve the loop bound by observing that the `INPUT` variable is invariant in the loop, although it might assume any value in the range [10..20] in the loop. Therefore, it does not contribute to the calculated loop bound. Thus, a safe upper loop bound, given that the loop terminates, is given by the number of values of `i` at point `p`: $\texttt{size(i,p)} = 20$.

Note that the derived loop bound is input dependent, i.e., another value limitation of the `INPUT` variable could result in a different loop bound.

4 Method details

This section will present our loop bound analysis and its included analyses in more detail. We also present some implementation details, useful for obtaining a faster and more precise analysis.

4.1 Program slicing

Our approach for loop bound analysis uses *program slicing* [18]. Program slicing finds a subset of a program containing the program parts which can affect some given part of the program like a specific condition, or a set of conditions.

Our program slicing works by first building a program dependency graph (PDG) which holds the *data flow* and *control* dependencies between the statements in the program [7, 13]. The slice which is computed with respect to some program part is the part of the PDG which is backwards reachable from the program part. For more details on our program slicing, see [16].

We slice w.r.t. to the exit conditions of the loop to be analyzed. Only the computed slice has to be analysed. In order to reduce the size of the program states of subsequent analyses we also remove variables that are not accessed.

Step-wise slicing Our current implementation actually performs a *stepwise slicing*. First, the program is sliced w.r.t. all conditionals in the program. This removes code that can never affect the outcome of any condition. Then the computed slice is sliced w.r.t. the exit conditions of each single loop to be analysed. Compared to slicing the original program for each loop, this two-step approach gives much better performance, especially if the first slicing is able to remove many statements and variables.

4.2 Abstract interpretation

Abstract interpretation (AI) is a theory of sound approximation of the semantics of computer programs. It was formalized by Cousot & Cousot [2].

AI gives a safe, but potentially pessimistic, estimation of the possible sets of states in different program points. To achieve this, abstract domains with elements representing sets of states, so-called "abstract states" are used. The abstract domains are complete lattices, with a top and a bottom value. For each statement in the language, a corresponding *transfer function* is derived which maps abstract states to abstract states. The transfer functions are used to set up a set of equations relating the abstract states for the different program points. An initial abstract state specifies possible constraints on the input variables. The set of equations is solved using least fixed-point iteration. The least fixed-point defines an abstract state for each program point, and each abstract state represents a safe overapproximation of the set of states in its program point. Often, the abstract states are mappings from program variables to *abstract values* representing possible sets of "concrete" values held by the variables.

For certain abstract domains, the fixed-point iteration will not always terminate. Termination can however be guaranteed through a binary *widening operator* on abstract states, which will enlarge the abstract states during the iteration [2]. Widening can also be used to speed up termination. The solution obtained using widening will be safe, but maybe not the least one. It can sometimes be improved using a *narrowing operator* [3].

Supported abstract domains Our current implementation supports two abstract domains, namely the *interval-* [10] and the *congruence* domain [1, 8]. It also supports the *product* domain of these two domains.

In the interval domain the possible values of a variable is approximated by an interval $[l..u]$. E.g., an abstract state holding the assignment `i` $= [1..20]$ represents all concrete states where $1 \leq$ `i` ≤ 20, i.e., 20 different states.

In the congruence domain the possible values of a variable is approximated by an abstract value of the form $n(\text{mod } m)$. For example, an abstract state holding the assignment `i` $= 0(\text{mod } 5)$ represent all concrete states where `i` contains the factor 5.

The abstract values in the product domain are pairs $\langle i, c\rangle$ where i is an interval and c a congruence. The pair $\langle i, c\rangle$ represents the intersection of i and c. For instance, $\langle [1..20], 0(\text{mod } 5)\rangle$ represents $[1..20] \cap 0(\text{mod } 5) = \{5, 10, 15, 20\}$.

In our implementation each basic data type in C, such as `char`, `int` and `float` has a corresponding abstract data type. We also have abstract versions of aggregate data structures, such a structs and arrays, as well as pointers. Our abstract domains model fixed-size integers with possible overflow. To guarantee termination of the AI we have implemented widening and narrowing operations for our different abstract domains. For details, see [10].

Figure 2 gives an illustrative example of the benefit of using the product domain. An interval analysis would derive `i` $= [0..9]$ at point `p`, corresponding to 10 concrete values. Similarly, a congruence analysis would derive `i` $= 0(\text{mod } 2)$ at point `p` corresponding to an infinite number of concrete values. However, the intersection of the two domains contains all values between 0 and 9 evenly dividable by 2, i.e., $\{0, 2, 4, 6, 8\}$. This set has the size 5, which is a precise loop bound.

```
1.   int i = 0;
2.   while(i < 10) {
3.       // p
4.       i += 2;
5.   }
```

Interval analysis:
$i = [0..9]$ at p

Congruence analysis:
$i = 0 (\text{mod } 2)$ at p

Figure 2. Interval and congruence example

4.3 Loop bound calculation

We use the result of the AI analysis of the sliced program to derive a loop bound for the selected loop. We first select a program point guaranteed to be within the loop, which all iterations of the loop are guaranteed to pass, e.g., the program point just before the last instruction in the loop header node[1]. Then, for all variables not ruled out by the analyses in Section 4.4, the sizes of their respective abstract values are taken as upper bounds to their numbers of possible concrete values. The loop bound is finally calculated by multiplying all these sizes.

For integer and pointer variables, the size of the set of concrete values defined by an interval, or an element in the product domain, is straightforward to compute. The same holds for aggregate objects (array, struct) containing only fields of integer and pointer type. If the variable is or contains a floating-point value then we consider the number of concrete values to be either zero, one or infinite.

If any variable in the abstract state holds the top value, then the loop bound cannot be derived (we consider top to represent an infinite set of concrete values). Similarly, if some variable in the abstract state holds the bottom value, then the loop body is unreachable and we set the loop bound to zero.

4.4 Invariant analysis

Invariant analysis is a program analysis used in many compilers [15]. It identifies statements in loops which can be moved outside the loop since they always recompute the same value. We have implemented a simplified version, which simply checks if any variable used in the (sliced) loop body is also possibly written in the body. A variable that cannot be written is considered loop invariant and can safely be excluded from the loop bounds calculation. Statements reachable through function calls must also be considered. Since pointers can be used to update values, e.g., in `int* p = &i; *p = 5;` variable `i` is assigned a value through the pointer `p`, we use the result of a pointer analysis to find which variables that could possibly be updated through dereferenced pointers.

[1] Assuming, for simplicity, that the loop is well-structured.

```
int i = 1;
while(i <= 100) {
    j = 1;
    while(j <= i) // p
       j++;
    i++;
}
```

(a) Nested loops

```
int temp;  // no init
int j = 0;
while(j < 100) {
    temp = 1;
    j = j + temp;
    temp = 2;
}
```

(b) Problematic code

Figure 3. Invariant analysis examples

Figure 3(a) gives an example where the invariant analysis helps producing a tighter loop bound. At program point `p` an AI using intervals would derive $i = [1..100]$ and $j = [1..100]$. This gives a loop bound of $100 * 100 = 10000$ of the inner loop. However, `i` is invariant in the inner loop, giving that the loop bound can be calculated using `j`'s abstract value only. This gives a loop bound of 100 for the inner loop.

Single-valued-uses analysis The condition detected by the invariant analysis, that a variable never is assigned a new value in the loop body, is unnecessarily strong. Actually, to remove the variable from the loop bounds calculation it suffices that *in any program point in the loop body where the variable might be used, it can hold at most a single value for a given execution of the loop.* An example is shown in Figure 3(b): here, an invariant analysis will fail since `temp` is reassigned two times in the loop, and yet it will always have the single value 1 when used.

We have implemented an analysis to discover if a variable only can have a single value at each relevant use. The analysis simply uses the abstract value derived by the AI, for each relevant variable and program point within the sliced loop body where it is used, to see if the variable can only hold a single value in that point. If this is true for all these program points the variable is removed from the loop bounds calculation.

5 The SWEET tool

SWEET (SWEdish Execution time Tool) [5, 10] is a research tool developed at Mälardalen University [14]. SWEET can handle ANSI-C programs including pointers, unstructured code, and recursion. The basic analysis steps of SWEET are depicted in Figure 4.

Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs its flow analysis on the intermediate representation (IR) of the compiler. The control structure of the IR and the object code is similar, and flow analysis results for the IR, in terms of execution bounds on basic blocks, is therefore also applicable for the object code. The low-level analysis of SWEET currently supports the NECV850E and

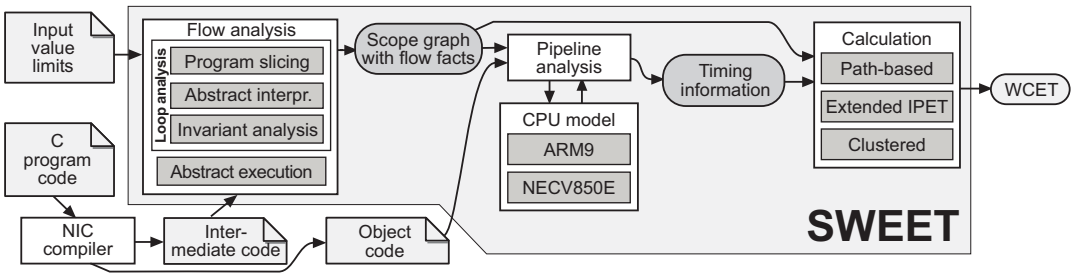


Figure 4. The SWEET WCET analysis tool

ARM9 processors. SWEET supports three different calculation methods: a path-based method, an IPET method, and a hybrid clustered method [5].

The loop bound analysis presented in this article is one of several analyses performed in the flow analysis phase. There is an annotation language which can be used to assign abstract values in the interval domain.

6 Measurements and evaluations

We have used programs from the Mälardalen WCET Benchmark suite [14] to test our flow analyses. The benchmarks are a diverse collection of test programs differing in types of flows, code structure and instructions, intended to thoroughly test different aspects of WCET analysis including flow analysis. Our current implementation of the loop bound analysis cannot handle recursive code, due to limitations in our AI. Thus, no recursive program has been used in our evaluations.

Table 1 gives some basic data about the programs, including lines of C code (**#LC**) and the number of loops (**#L**). The number of loops is counted in a context dependent manner since a loop might have different upper bounds depending from where its corresponding function is called, e.g., `crc` contains such an input dependent loop. For each benchmark we give the number (**#B**) and the percentage (**%B**) of loops bound by the analysis. We also give the number (**#E**) and the percentage (**%E**) of loops which are exactly bound, i.e., given a bound equal to the actual loop bound. The column (**Time**) gives the analysis time in seconds on a 3 GHz PC running Linux.

The **Total** row summarizes our analysis results. We see that more than 60% of all loops gets upper bounded and more than 50% are given an exact loop bound. The loops bounded are in most cases rather simple loops usually dependent on one or two integer index variables. For more complex loops, or loops containing floating point index variables, the analysis often fails.

The analysis time of the loop bound analysis depends very much on how much of the program that could be removed by the slicing. A large remaining program means that the AI usually will take quite a long time. For programs which take long time to analyse, like `adpcm`, `ns`, and `ludcmp`, the analysis time is dominated by the AI.

The results in the table are based on analysis using the interval domain. If we use the product domain described in Section 4.2, we are able to get tighter loop bounds for 6 loops.

7 Conclusions and future work

We have presented a static loop bound analysis based on a combination of standard program analysis techniques. The method has shown to be powerful, giving exact loop bounds for more than 50% of our used benchmarks, with reasonable analysis time.

For future work, we plan to extend the approach to handle more type of loops. One idea is to look into more powerful relational abstract domains in the AI, allowing constraints between values of variables. This should allow the size of the abstract states used for loop bound analysis to be minimized.

We plan to extend the approach to discover if a loop terminates. For example, if it can be shown that each loop variable is either monotonically increasing or decreasing, that no wrap-arounds of these variables could occur, that for any iteration of the loop at least one of the loop variables is updated, and we have a bound on the number of concrete states in the loop, the loop should terminate. We also plan to extend the approach to derive infeasible path information, i.e., paths never possible to execute within a loop body. The latter will be a combination of program slicing, AI and AE.

References

[1] S. Bygde. Abstract interpretation and abstract domains. Master's thesis, Mälardalen University, June 2006.

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

[3] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpre-

Program	Description	#LC	#L	#B	%B	#E	%E	Time
adpcm	Adaptive pulse code modulation algorithm.	879	27	18	67%	8	30%	48.6
bs	Binary search in an array of 15 integer elements.	114	1	0	0%	0	0%	0.81
cnt	Counts non-negative numbers in a matrix.	267	4	4	100%	4	100%	0.24
cover	Program for testing many paths.	640	3	3	100%	3	100%	0.32
crc	Cyclic redundancy check computation on 40 data bytes.	128	6	6	100%	6	100%	0.11
duff	Using "Duff's device" to copy 43 byte array.	86	2	1	50%	1	50%	0.04
edn	Finite Impulse Response (FIR) filter calculations.	285	12	12	100%	9	75%	0.71
expint	Series expansion computing an exponential integral.	157	3	3	100%	3	100%	0.04
fac	Recursive program to calculate factorials.	21	1	1	100%	1	100%	0.01
fdct	Fast Discrete Cosine Transform.	239	2	2	100%	2	100%	0.05
fft1	Fast Fourier Transform using Cooly-Turkey algorithm.	219	30	7	23%	3	10%	5.39
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	1	1	100%	1	100%	0.01
fir	Finite impulse response filter (signal processing).	276	2	2	100%	1	50%	0.38
inssort	Insertion sort on a reversed array of size 10.	92	2	1	50%	1	50%	0.54
jcomplex	Nested loop program.	64	2	0	0%	0	0%	0.04
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	375	3	3	100%	3	100%	0.06
lcdnum	Read ten values, output half to LCD.	64	1	1	100%	1	100%	0.01
ludcmp	LU decomposition algorithm.	147	11	6	55%	5	45%	247.6
matmult	Matrix multiplication of two 20x20 matrices.	163	7	7	100%	7	100%	0.51
ndes	Embedded code with many complex bit operations.	231	12	12	100%	12	100%	3.11
ns	Search in a multi-dimensional array.	535	4	1	25%	1	25%	91.9
nsichneu	Simulates an extended Petri net.	4253	1	1	100%	1	100%	1.11
prime	Search in a multi-dimensional array.	535	2	0	0%	0	0%	0.05
qsort-exam	Linear equations by LU decomposition.	121	6	0	0%	0	0%	76.4
qurt	Root computation of quadratic equations.	166	3	1	33%	1	33%	0.09
select	Selects the n:th largest number in floating point array.	114	4	0	0%	0	0%	19.6
statemate	Automatic generated code.	1276	1	0	0%	0	0%	1.00
ud	Linear equations by LU decomposition.	161	11	11	100%	10	91%	0.53
Total		-	164	104	63%	84	51%	-

Table 1. Benchmark programs and result of loop bound analysis

tation. In *Proc. 4^{th} International Symposium on Programming Languages, Implementations, Logics, and Programs*, Lecture Notes in Computer Science (LNCS) 631, pages 269–295. Springer-Verlag, August 1992.

[4] C. Cullmann. Statische berechnung sicherer schleifengrenzen auf maschinencode. Master's thesis, Universität d. Saarlandes, 2006.

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

[6] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[9] J. Gustafsson and A. Ermedahl. Experiences from applying WCET analysis in industrial settings. In *The 10th IEEE International Symposium on Oject/component/service-oriented Real-time distributed Computing (ISORC2007)*, Santorini Island, Greece, May 2007.

[10] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10^{th} IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Feb. 2005.

[11] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS'06)*, Dec. 2006.

[12] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2-3):129–156, May 2000.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

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

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[16] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET flow analysis by program slicing. In *Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06)*, pages 103–112, June 2006.

[17] Tidorum. Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t`.

[18] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

Analysing Switch-Case Tables by Partial Evaluation

Niklas Holsti

Tidorum Ltd

Tiirasaarentie 32, FI 00200 Helsinki, Finland

niklas.holsti@tidorum.fi

Abstract

Tracing the flow of control in code generated from switch-case statements is difficult for static program analysis tools when the code contains jumps to dynamically computed target addresses. Analytical methods such as abstract interpretation using integer intervals can work for some forms of switch-case code, for example a jump via a table of addresses indexed 1 .. n, but fail when the target compiler encodes the switch-case structure in a ROM table with a complex format and uses a library routine to interpret the table at run-time.

This paper shows how to extract the flow of control from such switch-case tables by partial evaluation of the table-interpreting routine. The resulting control-flow graph allows accurate analysis of the execution time and the logical conditions for reaching each case in the switch-case statement.

The method is implemented in Tidorum's Bound-T tool for worst-case execution-time analysis. The implementation builds on some basic Bound-T features for modeling program states in the flow-graph and propagating constant values through the graph.

1. Introduction

Static analysis of the worst-case execution time (WCET) of a program usually begins by building the control-flow graph (CFG). On the machine code level, where most WCET tools work, the tool has to find the possible successor instructions of each instruction under analysis. This is easy when the instruction defines its successors statically but hard for control-transfer instructions with dynamic target addresses, for example register-indirect jumps. Such *dynamic transfer of control* (DTC) instructions often result from switch-case statements [1, 6, 8].

The switch-case statement in languages such as C or Ada is a very flexible control structure. The programmer can choose the type of the switch index, for example an 8-bit or a 32-bit number; whether the cases are numbered densely 1 .. *n* or are a sparse subset of a large range; whether each case is reached by a unique index value or by a set or range of values; and whether there is a default case or not. Compilers often generate quite different kinds of code to implement different kinds of switch-case statements.

For small target processors such as the Intel 8051 or Atmel AVR some compilers try to reduce code size by encoding the switch-case statement into a ROM *switch table* and generating a call or jump to a *switch handler* routine that interprets the table at run-time. There may be several types of switch table, for example depending on the index type, each with its own switch handler.

This paper describes a way to find the full control-flow graph for code that uses switch tables and switch handlers. Section 2 defines a particular switch-table structure and the corresponding switch handler for use in examples. Section 3 states the problem and the goals for the solution. Section 4 defines the suggested solution as a form of partial evaluation. Sections 5 and 6 explain how this solution is implemented in the Bound-T WCET tool [2] and section 7 shows an example. Section 8 summarises the analysis method. Sections 9 and 10 report experience from implementation and experiments, respectively. Section 11 discusses related earlier work and section 12 concludes the paper.

When discussing the Bound-T implementation I will use the term “flow-graph” instead of the usual “control-flow graph”. Section 5 explains why.

2. Example of switch table and handler

The switch-table structure in this example was chosen to make the switch handler brief but not trivial. The structure is not taken directly from any compiler that I know of but is similar to real switch tables for 8-bit processors. The structure assumes a type of Atmel AVR processor with a 16-bit program counter and at most 64 KB of program memory.

In this example a switch table is a sequence of *entries*. An entry represents a set of 8-bit switch-index values that lead to the same case in the switch-case statement. An entry consists of four octets: a *mask* octet, a *match* octet, and the low and high octets of the 16-bit *address* for the case to be taken when the bit-wise logical “and” of the switch index and the *mask* octet equals the *match* octet. The order of entries in the table is arbitrary but the last entry always has a zero *mask* and *match*. This

represents the default case if there is one, else fall-through to the statement after the switch-case.

Consider this C subprogram *foo*:

```
void foo (unsigned char k)
{
   switch (k) {
   case 4:
      <statements for k = 4>
   case 8: case 9: case 11:
      <statements for k = 8, 9 or 11>
   default:
      <statements for other values of k>
   }
}
```

The switch table for this switch-case statement is shown in Table 1 below. It has four entries for a total size of 16 octets. Note that the second entry matches both $k = 8$ and $k = 9$ because the mask value 254 masks the least significant bit of k.

In this example a switch-case statement is compiled into code that loads the switch index (the parameter k in *foo*) into register *r0* and calls the switch handler *SwHandler*. The switch table is placed in the program memory immediately after the call so that the return address points to the first table entry. *SwHandler* searches the table for an entry that matches the switch index, then jumps to the *address* of this entry.

SwHandler can be written in AVR assembly language [3] as shown in Listing 1 below. For later reference the left margin shows the assumed word address of the instruction (in hex). Semicolons start comments that extend to end of line.

Listing 2 below shows the AVR code for function *foo* including the code for the switch-case statement and the hex form of the switch table. Listing 2 assumes that k is passed to *foo* in register *r16* and some arbitrary amounts of code in the case branches.

Listing 1. Example switch handler

```
SwHandler:
      ; Switch-case handler. Entered by call with the
      ; switch index in r0 and the switch table in
      ; program memory after the call instruction. Exits
      ; to the chosen case. Changes r1, r2, and Z.

0100  pop   r30       ; low octet of table address
0101  pop   r31       ; high octet of table address
      ; Z = r31:r30 = word address of the switch table.
0102  add   r30,r30   ; Multiply Z by two to make
0103  adc   r31,r31   ; it an octet address for lpm.

loop:       ; Z points at the next switch table entry.
0104  lpm   r1,Z+     ; r1 := entry.mask
0105  lpm   r2,Z+     ; r2 := entry.match
0106  and   r1,r0     ; r1 := index and mask
0107  cp    r1,r2     ; compare to entry.match
0108  breq  found     ; branch if entry matches index
0109  adiw  Z,2       ; no match, point at next entry
010A  rjmp  loop      ; try next entry

found:      ; Entry matches. Z points at entry.address.
010B  lpm   r1,Z+     ; r1  := address low octet
010C  lpm   r31,Z     ; r31 := address high octet
010D  mov   r30,r1    ; Z   := whole address
010E  ijmp            ; DTC jump to address in Z.
```

Table 1. Example switch table

mask	*match*	*address points to:*
255	4	the code for the case $k = 4$
254	8	the code for the case $k = 8$, 9 or 11
255	11	the code for the case $k = 8$, 9 or 11
0	0	the code for the default case

Listing 2. Example switch-case statement code

```
foo:
0200  mov   r0,r16   ; r0 := k
0201  call  SwHandler
      ; The switch table consists of the following
      ; 16 octets, shown in hex:
0203  FF 04 0B 02    ; k = 4, address = 020B
0205  FE 08 1C 02    ; k = 8 or 9, address = 021C
0207  FF 0B 1C 02    ; k = 11, address = 021C
0209  00 00 24 02    ; default, address = 0224
020B  < code for the case k = 4 >
021C  < code for the case k = 8, 9 or 11 >
0224  < code for the default case >
0229  ret            ; return from foo.
```

3. Problem and goals

The problem is to find the full control-flow graph for machine code that uses switch tables and switch handlers, for example with the structure described in section 2 but of course not limited to that example. The machine code is given as a memory image that is a mixture of code and data, not clearly demarcated. The solution should:

- find all cases of all switch-case statements,
- *not* mix up different switch-case statements to create false paths in the flow-graph,
- produce the sequence of instructions that leads to each case, so that later steps in the analysis can find an accurate WCET for each case,
- connect each case with the corresponding values of the switch index, again for use in later analysis steps (for example to find bounds for a loop that is nested in a case and depends on the switch index),
- apply uniformly to several kinds of switch tables and handlers and be robust to changes in their structure as the compilers evolve.

The solution should also be easy to implement in the generic, processor-independent parts of a WCET tool, in my case Bound-T [2], with minimal changes to the processor-specific parts, for example the parts of Bound-T that decode AVR instructions.

Bound-T can analyse many aspects of a subprogram in a (calling-) context-dependent way but the flow-graph of a subprogram must be independent of context. Analysing a switch handler (for example *SwHandler*) as an ordinary, independent subprogram cannot give a context-dependent resolution of the DTC (the *ijmp* in *SwHandler*). Instead, a switch handler must be analysed as an integral part of the subprogram that contains the switch-case statement (for example *foo*). This is similar to in-line expansion of the call to the switch handler.

The target addresses for the DTC result from executing the switch-handler instructions that access the switch table. The analysis must thus simulate or execute these instructions. Furthermore, the analysis must unroll the table-scanning loop in the switch handler. Each iteration of the loop leads to a different case; unrolling the loop separates the paths to the different cases for separate analysis.

4. The solution by partial evaluation

Partial evaluation is the execution of a program with some inputs bound to concrete values but other inputs not so bound (free input variables) [4]. The result is therefore not a concrete output value but a *residual program* that still depends on the unbound inputs. The residual program is a *specialization* of the original program: it is specialized to the domain where the bound inputs have the given values.

The proposed analysis of switch tables and switch handlers uses partial evaluation of subprograms as follows. A switch handler is a subprogram with two inputs: the switch index and the switch table. At analysis time, in a given invocation of a switch handler for a given switch-case statement the switch index is usually unbound (has an unknown, dynamic value) but the switch table is bound to a static constant: the table generated for this switch-case statement.

If we partially evaluate the switch handler under this binding, the residual subprogram depends only on the switch index and not on the switch table. The partial evaluation resolves the DTC instructions into control transfers with static target addresses, copied or computed from the switch table.

For the switch handler shown in section 2 partial evaluation with a known switch table means that we know the value loaded by the execution of any *lpm* instruction. Thus the target address of each possible execution of the *ijmp* DTC instruction is known even if the value of the switch index (*r0*) is unknown.

Within the Bound-T tool the partial evaluation is implemented in a way that fits the Bound-T architecture, not as a general-purpose partial evaluator such as the *mix* evaluator described in [4]. In Bound-T the original, unevaluated subprogram (the switch handler) is represented implicitly by its entry address and the instructions in the target program that can be reached from the entry address. The residual subprogram (the switch handler specialized to a given switch table) is represented as a part of the flow-graph of the subprogram that contains the switch-case statement. This part is a subgraph rooted at the node that invokes the switch handler. The nodes of the subgraph represent (executions of) instructions in the switch handler; the leaves of the subgraph represent the DTC leading to each case.

In the terminology of [4] the *source language* of this partial evaluator is machine-code program-memory images and the *target language* is Bound-T flow-graphs. (As a part of Bound-T the *implementation language* is Ada, but this is not important.)

The next two sections explain how partial evaluation is implemented in Bound-T and why it is a natural extension of the way in which Bound-T builds flow-graphs from machine code. This says more about Bound-T than about the partial evaluation method for switch-case analysis. Eager readers may skip to section 7 for an example of the analysis.

5. Building flow-graphs in Bound-T

This section describes the structure of flow-graphs in Bound-T and the iterative algorithm for building flow-graphs from machine code. The next section extends the algorithm to include partial evaluation.

First a definition of terms. The internal representation of a subprogram in Bound-T is a *flow-graph* (FG). A flow-graph differs from a control-flow graph (CFG) because (as defined in this paper) a CFG node represents a given machine instruction in *any* program state while an FG node represents a given instruction in some *subset* of program states. Thus, a given instruction is always represented in at most one CFG node, but can be represented in several FG nodes when this instruction is modeled separately for different program states. Bound-T adopted the flow-graph concept to model complex control mechanisms such as nested zero-overhead loops in DSPs.

The abstraction of the program state that is used for flow-graphs is called the *flow-state*. The program counter (PC) is always a concrete part of the flow-state; a flow-state implies a PC value. Each node in a flow-graph is *tagged* with a flow-state. No other node in this flow-graph is tagged with this flow-state.

Each flow-graph node has several attributes to model the instruction in this node. For this paper the main attribute is the *computational effect*: a set of assignments of expressions to variables (registers or memory locations). For example, the effect of the AVR instruction *lpm r1, Z+* is modeled by the assignments $r1 := pm[Z]$, $Z := Z+1$ where $pm[Z]$ stands for the value of the program memory octet at address *Z*. (Ignore the fact that the 16-bit *Z* pointer is composed of the two 8-bit registers *r30* and *r31*. This complication is nasty but not relevant here.)

Each edge in the flow-graph is provided with a Boolean expression that is a necessary but perhaps

not sufficient *condition* for taking this edge. For example, the condition for the branch-taken edge after the AVR instruction *breq* is that the "zero" flag be set, here written as $zf = 1$. The condition is evaluated after the effect of the source node.

Bound-T starts the analysis of a subprogram by building the flow-graph of the subprogram. This is an iterative algorithm very like the algorithm in [6]. For each new flow-state the algorithm first adds a blank node to the flow-graph and then proceeds to fill in the blank nodes with their attributes. The final flow-graph contains all flow-states and instructions in the subprogram that can be reached from the entry address. The algorithm follows.

Building the flow-graph of a subprogram in Bound-T
Initialization. The flow-graph is initialized to consist of one blank node tagged with the flow-state that represents the entry address of the subprogram.
Iteration. The algorithm repeatedly executes the *Fill node* step until there are no blank nodes in the flow-graph.
Fill node. Pick a blank node *N* from the flow-graph. The flow-state of the node identifies (through its PC value) the instruction executed in this state. Fetch this instruction from the memory image of the target program and fill in the attributes of node *N* from this instruction. Determine all successor flow-states for node *N*. The successor of a normal call instruction is the return point in the caller. A normal return instruction has no successors. For each successor state *s* of *N*, if there is not already a flow-graph node *S* tagged with *s* then add such a new blank node *S* to the flow-graph. Make a new edge from *N* to *S*.

The analysis of normal subprogram calls in Bound-T is not relevant to this paper because a call to a switch handler will be analysed as if the call were *in-lined*. A switch-handler call is analysed as a kind of jump instruction by a simple variation of the above algorithm: the successor of a call to a switch handler is taken to be the first instruction in the switch handler, instead of the return point in the callee.

For the example in section 2 the return point contains the switch table, not AVR instructions, so control never reaches the return point.

6. Partial evaluation in Bound-T

This section explains how the flow-state concept was extended to implement partial evaluation during flow-graph building in Bound-T.

First note that the flow-graph building algorithm can already be viewed as partial evaluation. The entry address (initial PC value) is one input for the target program; building the flow-graph amounts to partial evaluation of the target program with respect to this input, keeping all other inputs unbound. The partial evaluation of an instruction amounts to finding the effect of the instruction on the PC, in other words finding the successor instructions.

We can extend this partial evaluation simply by *adding more concrete state components to the flow-state*. Of course, this forces us to *compute the effect of each instruction on these new flow-state components* to find the successor flow-states of the instruction.

To implement this in Bound-T the flow-state type is extended with a *data-state* component that is either null or a pointer to a *data-state object*. A data-state object models the values of program variables just before executing the node tagged with this data-state.

For this paper a data-state object is a *partial mapping of variables to values*. In other words a data-state binds some variables to known values but leaves all other variables unbound. For example, the data-state on entry to *SwHandler* from the call in *foo* could bind the variable holding the return address (the top stack word) to the value 0203 (hex) and leave all other variables unbound.

The flow-graph building algorithm is extended to handle data-states as follows. When partial evaluation is not in progress the data-state is null and the algorithm works as before. Otherwise the algorithm uses the data-state to partially evaluate the computational effects and edge conditions and uses the residual effects and conditions to update and propagate the data-state over nodes and edges.

Handling data-states while building the flow-graph
When filling a node for a given flow-state. If the given flow-state has a non-null data-state, partially evaluate the computational effect of the node on this data-state and store the *residual* effect in the node. Also create the *post-state* of the node as the given data-state updated by the residual effect: assignment of a constant binds the target variable, other assignments unbind it. The post-state models the program state *after* executing the instruction in this node. If the given flow-state has a null data-state make the post-state null too.
When adding an edge from a source node to a successor flow-state. If the post-state of the source node has a non-null data-state, partially evaluate the given edge condition on this data-state and if the result is *false* discard the edge as infeasible. Otherwise store the *residual* condition in the edge. If the successor flow-state has a specified data-state (whether null or not) use it as such (this happens when starting or stopping partial evaluation). Otherwise use the post-state of the source node but constrained by the residual edge condition: if the condition implies a known value for a variable then update the successor data-state with this binding.
When filling a DTC node. If the node has a non-null data-state then try to compute the target address from the data-state. If this succeeds (*ie*. if the DTC target depends only on variables bound to constants in the data-state) then add the corresponding (static) edge; also, if this DTC represents an exit from a switch handler then put a null data-state in the target of the new edge, to stop partial evaluation on this path.

The extensions to the algorithm use existing Bound-T services for propagating constant values in flow-

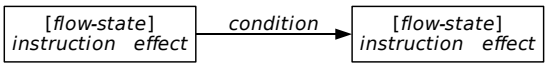
graphs and computational effects. New code was needed mainly for the container of data-state objects.

Most of the extensions for data-state handling are implemented in the processor-independent parts of Bound-T. The processor-specific modules only have to start and stop the partial evaluation at suitable points in the analysis. For this paper I assume that the processor-specific modules detect when a call or jump instruction enters a switch handler; at that point these modules start partial evaluation by putting the initial data-state for the switch handler in the target of the edge that enters the switch handler. Likewise, I assume that processor-specific modules detect when a DTC is an exit from a switch handler. Section 8 discusses these assumptions.

7. Example

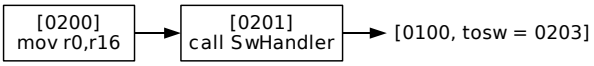
This section shows how the partial evaluation works for the *foo* function and the *SwHandler* from section 2 by a series of snapshots of the growing flow-graph.

Nodes and edges in the flow-graph are drawn as follows:



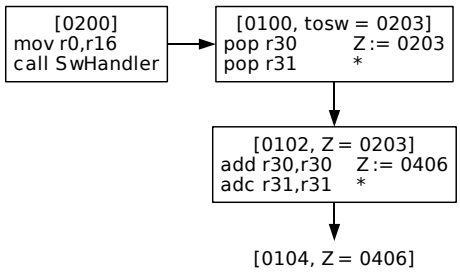
The flow-state is shown in brackets [] at the top of the box that depicts a node. The flow-state starts with the instruction address (PC) in hex, followed by the data-state bindings if any. For brevity only relevant bindings are shown. The AVR instruction is shown below the flow-state, followed by the relevant parts of its residual computational effect. An edge with no condition is unconditional (always taken). Blank nodes are shown as a bare "[*flow-state*]" with no box.

The first figure below shows the flow-graph of *foo* after the first two instructions are inserted (with null data-states) and just after detecting that the second instruction (the call) enters a switch handler. The AVR-specific modules of Bound-T have accordingly defined the successor of the call to be the first instruction in *SwHandler* (PC = 0100 hex) with a data-state that binds the return address (top of stack word, *tosw*) to 0203 hex. There is one blank node with this flow-state [0100, *tosw* = 0203].



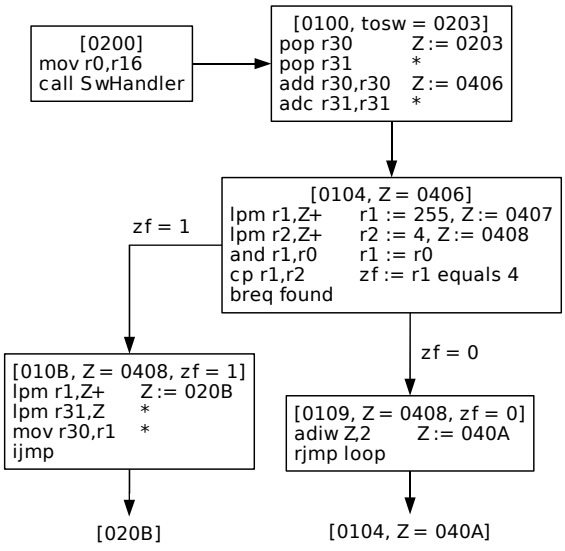
As the flow-graph grows I will compress the figures by showing several successive instructions in one box.

The next figure shows the flow-graph when the first four instructions from *SwHandler* have been inserted. Note how the partial evaluation of the *pop* instructions transformed the *tosw* binding into a binding for the *Z* pointer and how the evaluation of the *add* and *adc* instructions doubled the value bound to *Z*. (The asterisks indicate a computational effect that was combined with a preceding instruction to build a 16-bit operation from two or more 8-bit operations.) The single blank node shows that the next instruction to be added is the first instruction in the loop in *SwHandler*, at address 0104, with a data-state binding *Z* to 0406 hex, the octet address of the first entry in the switch table.



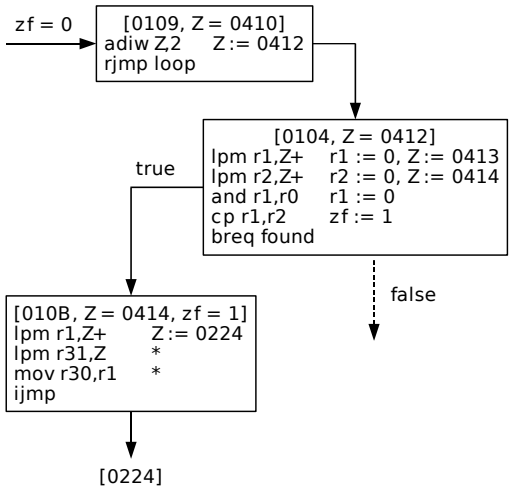
The next figure shows the flow-graph when it contains all the loop instructions and the first possible exit from the loop (for $k = 4$). On the left the loop exits when *zf* = 1. The *ijmp* DTC is resolved to a static jump because the data-state binds *Z* to 020B hex. This identifies the first case of the switch. Partial evaluation in this branch stops because the successor flow-state [020B] has a null data-state.

On the right, when *zf* = 0, the loop is about to repeat (*rjmp loop*). The successor flow-state contains the address of the loop-head (0104) which is already represented by a filled node, but it has a different data-state: *Z* is bound to 040A, not 0406 as in the existing node. The algorithm therefore creates new nodes for the second iteration of the loop. In fact the loop will be fully unrolled because the data-state binds *Z* to a different value in each iteration of the loop.



The third loop iteration is unrolled in the same way. The figure below shows the flow-graph parts for the fourth iteration which accesses the last switch-table entry (the default case) at octet address 0412 (word address 0209). The loop-repeating edge with the original condition *zf* = 0 becomes infeasible because the data-state binds *zf* to 1, making the residual con-

dition *false*. This ends the unrolling and also the partial evaluation.



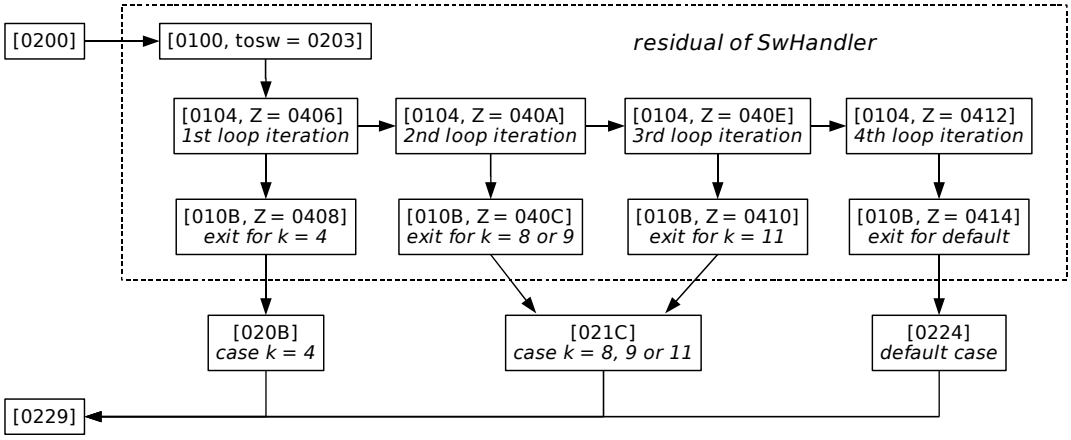
The last and largest figure, below, is an overview of the final flow-graph of *foo*. The residual form of *SwHandler* within this flow-graph is a tree of comparisons and conditional branches that explicitly models the sequence of instructions leading to each case. The edge conditions (not shown in the figure) define the corresponding index values.

8. Summary

To summarise, this method for finding the flow of control encoded in switch tables comes in three parts:

1. A flow-graph structure that can model the same instruction separately in different states (the flow-state in Bound-T).
2. A state abstraction and transfer functions for data values (the data-state and computational effects in Bound-T).
3. Means to detect entry to and exit from a switch handler in order to start/stop partial evaluation.

The first two points enable partial evaluation of machine code into parts of flow-graphs. The third point applies partial evaluation to reveal the flow of control in switch tables.



This partial-evaluation method largely achieves the goals listed in section 3. The two main problems left are processor-specific. The first problem is to model the computational effects of all instructions so exactly that the partial evaluation of the switch handler resolves the DTCs. For example, no version of Bound-T now models the "half carry" flag for BCD arithmetic. If a switch handler uses this flag in a DTC the partial evaluation will not resolve the DTC.

The second problem is to detect when a switch handler is entered or exited. Bound-T now uses the compiler-specific identifiers of the switch handlers and works only if these identifiers are present in the symbol-table of the program. An alternative could be to use data-flow analysis or slicing as in [5-9] to detect that a given DTC is "table driven".

Other applications of partial evaluation in WCET analysis can be imagined. For example, the *printf* function in C is notoriously difficult for WCET analysis because it is extremely data-dependent; *printf* is really an interpreter driven by the contents of the format string. Now, the great majority of *printf* calls have a constant string as the format parameter, for example:

```
printf ("%d and %f\n", ivar, fvar);
```

Partial evaluation of such a *printf* call with respect to the constant format string should transform the interpretive loop over the format string into sequential code in the residual flow-graph. It should also transform most format-dependent conditional branches into unconditional flow of control. In the above example, the residual flow-graph should contain one *%d* (decimal integer) formatting action, followed by formatting of the constant string `" and "`, followed by one *%f* (decimal floating-point) formatting action, followed by a new-line

action. Thus, partial evaluation should resolve the format-dependent aspect of the WCET for *printf*.

In general, partial evaluation could help the context-specific analysis of any subprogram that has control-flow that strongly depends on a parameter that is often constant, or that can be resolved to a constant by other analysis.

9. Implementation experience

So far Tidorum has implemented this method of switch-case analysis in Bound-T for two target processors: Atmel AVR and Intel 8051. The compilers currently supported are the IAR and Keil compilers for the 8051 and the IAR compiler for the AVR. The method works as expected but the implementation of course showed that some extensions were necessary. This section briefly describes the extensions.

Firstly, on the AVR processor some switch handlers use single-bit load and store instructions that work with the dedicated "T" bit in the status register. As foreseen in section 8 it was necessary to extend Bound-T/AVR with models for this bit and these instructions, in order to get good residual branch conditions and to terminate the partial evaluation.

Secondly, some switch handlers call their own subroutines, for example to load the next entry from the switch table into registers, compare it to the switch index, and execute a DTC when they match. During partial evaluation all calls to these handler subroutines also have to be in-lined in the flow-graph of the subprogram that contains the switch-case statement.

Thirdly, some switch handlers implement DTC by pushing the target address on the processor stack and executing a return instruction as if the target address were a return address. Bound-T was extended to use data-flow analysis of the stack contents to separate such DTC "returns" from ordinary returns.

10. Experiments

Testing the method on several AVR and 8051 programs showed that the residual flow-graphs were less complex than could be feared. The switch handlers contain many conditional branches within loops and so loop unrolling could create quite large residual flow-graphs. However, the partial evaluation resolves many branch conditions to constants, leaving unconditional branches. This reduces the number of basic blocks in the residual flow-graphs (note that Bound-T allows unconditional branches within basic blocks) but leaves an unusually large number of instructions per basic block.

The IAR compiler for the AVR has an option that controls the kind of code generated for switch-case statements. This option can force the compiler to use in-line comparison code, or a switch table with a call to a shared switch handler, or a switch table with an in-lined form of the switch handler. Comparing the first two forms for some switch-case statements with 8-bit index values showed that the WCET for a switch handler call is usually much larger (by a factor of 10 or so) than the WCET for in-line comparison code. This is explained by the very general design of the IAR switch tables and switch handlers. The ratio would probably be less for multi-octet index-types where the in-line comparison code must be larger.

11. Related work

Earlier work on switch-case control flow [5-8] uses program slicing and constant propagation to find dense tables of addresses or jumps, indexed by the switch index. This is related to but not the same as partial evaluation. Only the "hashing form" in [8] involves run-time search in a table.

Bound-T analyses jumps through dense address tables with a combination of instruction-pattern matching, to find these jumps, and data-flow analysis (based on Presburger Arithmetic) to find the bounds of the address table. The instruction patterns in Bound-T are currently target-specific and inflexible; slicing methods and data-flow patterns as in [5-8] would be an improvement.

De Sutter *et al*. in [5] start from a "super-conservative" control-flow graph in which each DTC is modeled as an edge to a special, unique "hell node" that represents an unkown program point. The graph also has edges from the hell node to every node that could be the target of a DTC. Constant propagation then prunes away some of these hell edges. Kästner and Wilhelm present a similar top-down method [7]; however they identify address tables by their assembly-language form (lists of label identifiers), not by their usage.

The drawback of such top-down flow-graph pruning methods [5, 7] is that they first need some other method to *find* all the instructions in the program. That is impractical in the analysis of binaries for processors with instructions of different sizes, because the common executable-file formats such as ELF do not mark instruction boundaries in the memory images. Some cross-compilers even deliberately overlay instructions so that, for example, the code can jump to the second octet of a 3-octet instruction to use the last two octets as a 2-octet instruction. The bottom-up flow-graph extension methods in Bound-T and [6] work also for such processors and compilers.

Another problem with some top-down methods is that they assume that a subprogram consists of *contiguous* code. This is is false for several cross-compilers that implement optimizations such as shared subprogram epilogues where the code for a subprogram can jump into the middle of some other subprogram far away in the address space.

Tröger and Cifuentes [9] use slicing to find calls of virtual functions in the object code of C++ programs, another form of DTC. They model the computational effect of instructions in a "High-Level Register Transfer Language", HRTL, similar to the formulation in Bound-T. The static part of their analysis does not try to find the actual addresses of the callees. Instead,

an HRTL interpreter dynamically executes the program and records the computed target addresses of the virtual function calls. However, the interpreter records only the executed paths, not all possible paths as in partial evaluation, so the virtual function tables may not be fully explored.

I know of no other analysis tool that creates residual subprograms in flow-graph form. However, this is much like context-specific optimization while compiling an in-lined subprogram.

The "abstract execution" method in SWEET [10] creates graphs of data-states (execution histories) but does not expand the CFG. The current SWEET input language, NIC, does not use switch tables.

12. Conclusion

I see this switch-case analysis as a small example of cooperation between the two main approaches to program analysis, the first being concrete state *enumeration* by executing the program and the second being state *comprehension* by abstracting the program. Partial evaluation represents the first approach. Bound-T uses the second approach to find loop bounds. I believe that WCET analysis would benefit from more use of state enumeration. The problem, of course, is choosing which state components to enumerate. Enumerating the states in switch handlers was an easy instance of this problem.

References

[1] G. Bernat and N. Holsti. Compiler Support for WCET Analysis: a Wish List. In *Proc. of the 3rd International Workshop on WCET Analysis* (WCET 2003), Porto, July 2003.

[2] Tidorum Ltd. Bound-T Execution Time Analyzer. *http://www.bound-t.com*.

[3] Atmel Corporation. 8-bit AVR Instruction Set. Rev. 0856D-AVR-08/02.

[4] N.D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, Vol. 28, No. 3, September 1998, pp. 480-503.

[5] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2000, pp. 1013-1019.

[6] H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 23-30.

[7] D. Kästner and S. Wilhelm. Generic Control Flow Reconstruction from Assembly Code. *Proc. LCTES'02 – SCOPES'02*, June 2002, pp. 46-55.

[8] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proc. of the 7th International Workshop on Program Comprehension*, May 1999, pp. 192-199.

[9] J. Tröger and C. Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In *Proc. Ninth Working Conference on Reverse Engineering* (WCRE'02), 2002, pp. 65-74.

[10] J. Gustafsson, A. Ermedahl, C. Sandberg and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. of the 27th IEEE International Real-Time Systems Symposium* (RTSS'06), December 2006, pp. 57-66.

Analysis of path exclusion at the machine code level

Ingmar Stein and Florian Martin
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
{stein,florian}@absint.com, http://www.absint.com

Abstract

We present a method to find static path exclusions in a control flow graph in order to refine the WCET analysis. Using this information, some infeasible paths can be discarded during the ILP-based longest path analysis which helps to improve precision. The new analysis works at the assembly level and uses the Omega library to evaluate Presburger formulas.

1 Introduction

A commonly used method to calculate worst-case execution times (WCET) for a program is to maximize

$$t_G = \sum_{n \in N} c(n) \cdot t(n)$$

where $G = (N, E, s, x)$ is the control flow graph representing the program, $c(n)$ is the execution count of a basic block n and $t(n)$ is the runtime of n. This optimization problem can be formulated as an Integer Linear Program (ILP) and solved by widely available ILP solvers.

The result of an ILP-based path analysis is a path that represents a safe upper bound of the execution time. However, it is possible that this path can never occur at runtime. At a fork in the control-flow graph, the decision which of the successor nodes will be executed next often depends on the path that leads to the fork. Depending on the execution history, only one of two successors might be feasible. Those dependencies are not accounted for in the ILP, and the path analysis views both nodes as possible successors. This situation can lead to a drastic overestimation of the real WCET.

In this paper, we introduce an extension of the aiT [1] analyzer that incorporates those dependencies into the ILP, which in turn improves the WCET prediction. The analysis produces additional ILP constraints that can exclude several classes of infeasible paths.

The example in Figure 1 illustrates how flow facts can be beneficial for the WCET computation. In this example, the path analysis has to select the successor nodes with the highest costs for both of the branches A and D. The resulting WCET is the sum of the costs associated with the edges constituting the critical path, i.e. $100 + 100 = 200$.

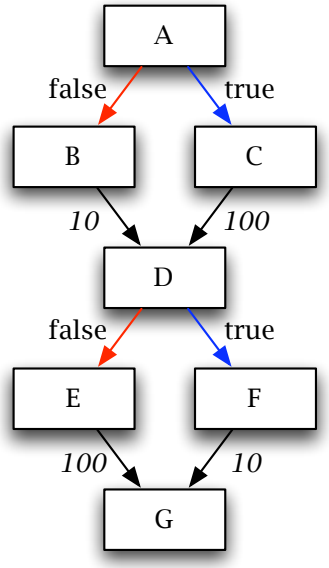


Figure 1. A control flow graph

However, if the analysis finds out that a positive outcome of the branch condition at A implies a positive outcome of the branch condition at D and vice versa, it creates a flow fact which allows only the paths $ACDFG$ and $ABDEG$. As a result, the new critical path has a WCET of $100+10 = 110$.

Such constructs as in the example often occur in code generated by code-generators such as SCADE [2] or in mode-driven code where many execution paths are controlled via relatively few flags.

2 Overview

The input for the flow constraint analysis is the control-flow graph of the program. While traversing this graph, each conditional branch is visited and an expression describing the branch condition is built. This step is trivial for high-level programming languages where the conditions are given in the source code, but as we are facing machine code, we have to reconstruct this information. Using a slicing component which operates on the assembly level, we find a set of instructions and variables that contribute to the branch conditions. If all instructions contained in that set can be mapped to arithmetic or comparison operations, we can build a boolean expression representing the branch condition.

In a second step, the expressions are transformed into another representation suitable for a solver library (Omega). The solver is used to compare two expressions, i.e. to check whether one expression implies the other or whether they are even equivalent. Beforehand, we test whether the two expressions can actually occur on the same path because not every implication allows for a sensible statement about the program.

The results of the comparisons are used to create new ILP constraints that are added to the ILP for the path analysis. This leads to a higher precision of the WCET prediction, i.e. a predicted worst-case execution time that is lower than the predicted WCET without the flow constraint analysis, but still is a safe upper bound of the real WCET.

3 The Flow Constraint Analysis

The flow constraint analysis traverses the control-flow graph and inspects all conditional branches, i.e. all inner nodes with more than one successor that are not call nodes.

If value analysis finds the exact (singleton) value of the condition register at a conditional branch, it marks one of the two outgoing edges as infeasible, and additional flow facts cannot improve the situation any more. Hence, only those branches where value analysis cannot deduce the value of the condition register are relevant for the flow-fact generation; the ones whose outcome is already determined by the value analysis are skipped.

A backward slice is computed for each considered conditional branch using the condition register as the initial target. A slice is a set of program points that directly or indirectly participate in the computation of the slicing criterion. A method how to compute slices is presented in [5].

Definition 3.1. A slice is called *linear* iff the program points contained in the slice can be ordered such that each program point is dominated by its predecessor. A linear slice that is ordered like that is called an *ordered slice*.

Example 3.1 (Linear slice)**.** Figure 2 shows two control-flow graphs. The instructions that constitute two different slices are highlighted using a bold border. The left graph represents a linear slice because the two basic blocks can be ordered as A, D and block A dominates block D. In contrast, the right graph is non-linear because block C dominates neither D nor A.

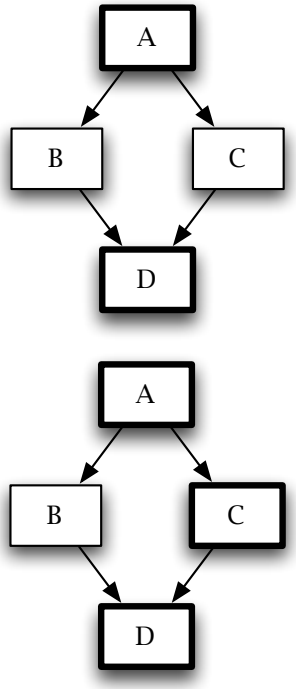


Figure 2. Linear slice (upper) and non-linear slice (lower)

We now restrict the analysis to linear slices. This excludes exactly those conditions that are built up on several different paths. The ordered slices are then transformed into slice trees. The inner nodes of a slice tree represent instructions while the leaves are either registers, memory cells, or constants (see for instance Figure 3).

Slice trees containing memory accesses whose target addresses cannot be determined statically cannot be used for the following comparisons and are therefore discarded.

A slice tree is an intermediate representation that can be transformed into other formats for different theorem provers. This process is described in the following for the Omega library.

The Omega Project is a collection of “Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs” by William Pugh and the Omega Project Team [4]. In particular, Omega offers a tautology test for

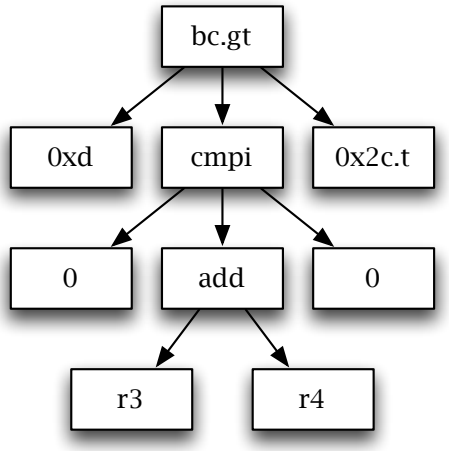


Figure 3. A slice tree

Presburger formulas that we will use to compare the branch expressions.

Definition 3.2. *Presburger arithmetic* is defined as an arithmetic with the constants 0 and 1, a function $+$, a relation $=$ and the axioms

1. $\forall x\colon\ \neg(0 = x + 1)$;
2. $\forall x\,\forall y\colon\ \neg(x = y) \implies \neg(x + 1 = y + 1)$;
3. $\forall x\colon\ x + 0 = x$;
4. $\forall x\,\forall y\colon\ (x + y) + 1 = x + (y + 1)$;
5. If $P(x)$ is a formula consisting of the constants $0, 1, +, =$ and a single free variable x, then the following formula is an axiom

$$(P(0) \wedge \forall x\colon\ P(x) \implies P(x{+}1)) \implies \forall x\colon\ P(x).$$

Presburger arithmetic is a decidable fragment of arithmetic and implementations of fully automatic decision procedures (such as Omega) are readily available.

Slice trees are translated into Omega trees by mapping the semantics of the individual instructions to arithmetic or comparison operations. Instructions with unknown semantics are treated as symbolic functions. Several patterns are used during the translation of instructions into Omega operators that allow for the combination of multiple instructions into a single operator. While the inner nodes of Omega trees represent operations, the leaves are translated as follows:

- Integer constants remain constants.
- Registers and memory cells become free variables. A prefix of the variable name encodes the type of the variable as shown in Table 1.

Prefix	Type	Suffix
r	Register	Register number
m	Memory cell (word)	Memory address
h	Memory cell (halfword)	Memory address
b	Memory cell (byte)	Memory address

Table 1. Omega tree leaves

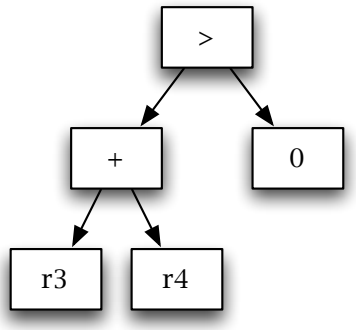


Figure 4. An Omega tree

Figure 4 shows the Omega tree resulting from the slice tree of Figure 3 using a simplified notation.

If all conditional branches are annotated with Omega trees, we can compare the branch conditions of two basic blocks A and B by testing several boolean expressions using Omega: $A \implies B$, $A \implies \neg B$, $\neg A \implies B$, $\neg A \implies \neg B$ and the same expressions with A and B swapped. If Omega determines one of the expressions to be a tautology, we can derive the flow constraints according to Table 2. The names a_t, a_f, b_t, and b_f stand for the *true* and *false* successors of the two basic blocks a and b, and $c(x)$ the execution count of basic block x. The table includes expressions that are logically equivalent to cover those cases where some of the successors a_t, a_f, b_t, and b_f are unavailable.

Expression	Flow constraint
$A \implies B$	$c(a_t) \le c(b_t)$
$A \implies \neg B$	$c(a_t) \le c(b_f)$
$\neg A \implies B$	$c(a_f) \le c(b_t)$
$\neg A \implies \neg B$	$c(a_f) \le c(b_f)$
$B \implies A$	$c(b_t) \le c(a_t)$
$B \implies \neg A$	$c(b_t) \le c(a_f)$
$\neg B \implies A$	$c(b_f) \le c(a_t)$
$\neg B \implies \neg A$	$c(b_f) \le c(a_f)$

Table 2. Implications and corresponding flow constraints

4 Limitation to n bits

Omega operates on the domain of integers, therefore the variables in the Presburger formulas have no range restrictions. However, the machine arithmetic works on n bits and is thus not modelled correctly in the Omega expressions. To resolve this problem, one can introduce modulo operators in the expressions to simulate an n bit range. If C is an expression whose result is an n bit value, C is replaced by $C' = C \bmod 2^n$. Because Presburger expressions don't have a built-in modulo operator, another substitution is needed:
If a term $x \bmod c$ occurs in a constraint C', C' is replaced by

$$\exists\gamma : c\gamma \leq x < c(\gamma + 1) \wedge C''$$

where C'' is derived from C' by replacing $x \bmod c$ by $x - c\gamma$.

5 Evaluation

In order to evaluate the effectiveness of the analysis, we have analyzed a set of test programs. All tests were performed using aiT for MPC755. Table 3 illustrates how the WCET changes if path analysis is run without or with the flow constraints (WCET_{fc}). The last column shows the number of generated flow facts. The runtime of the flow constraint analysis on the test programs is presented in figure 5.

6 Outlook

With the main work done, we now look at possible future enhancements and additional uses of the flow constraint analysis.

6.1 Portability.

We plan to implement the analysis for further microarchitectures besides the PowerPC platform. The ARM platform is a natural extension since the slicing component already exists for it.

6.2 Nonlinear slices.

Furthermore, it seems worthwhile to examine nonlinear slices to find out whether new opportunities for optimization arise if the linearity constraint is dropped. Nonlinear slices may be handled by using a data-flow analysis that propagates the node conditions and subsequently combines all conditions associated with a node. However, the risk is very high that the resulting expressions grow too large for the Omega library and that the runtime increases by several orders of magnitude.

6.3 Theorem-prover interface.

In addition to this, other theorem provers could be evaluated by providing an interface to the flow constraint analysis. An alternative prover could provide a performance superior to Omega in some cases or offer more functionality such as floating-point support.

6.4 Elimination of unreachable code.

With a simple extension, flow constraint analysis is able to detect some cases of unreachable code and to exclude the respective code blocks from the subsequent analyses, e.g., pipeline analysis. For that, a condition of a child node is compared to that of its direct parent. If they are equivalent or complementary, one of the two successors of the child node can be marked as infeasible.

6.5 PAG.

Unreachable code elimination as described above is an example how the information gathered by flow constraint analysis can be used for additional purposes. Another use case is PAG-generated analyzers [3] whose precision can be improved by path exclusions.

7 Conclusion

We have presented a method to find path implications within a given control flow graph for machine code programs. This information has been used to generate constraints which are then added to the ILP of the path analysis. The so-called flow constraints contribute to an improvement of the WCET prediction by excluding paths which cannot occur at runtime.

A tool which implements the algorithm presented in this paper has been successfully integrated into a WCET framework. It represents another phase in the workflow of the aiT WCET analyzer and fits seemlessly into the existing infrastructure. The tool has been used to conduct several tests which show both the effectiveness of the flow constraint analysis (ppcbrunch) on industrial programs as well as the moderate runtime increase of the complete WCET analysis as can be seen in figure 6.

References

[1] AbsInt Angewandte Informatik GmbH. ait: Worst-case execution time analyzers.

Program	WCET	$WCET_{fc}$	Improvement	Constraints
Synth. example 1	1440 cycles	1154 cycles	19.9 %	4
Synth. example 2	1140 cycles	819 cycles	28.2 %	5
avionic 1	1480 cycles	1420 cycles	4.1 %	1
avionic 2	3178 cycles	3050 cycles	4.0 %	8
zlib	6706 cycles	5242 cycles	21.8 %	2

Table 3. Results for several test programs

Program	Instructions	Basic Blocks	Size [Bytes]	Type
Synth. example 1	44	13	912	Mach-O
Synth. example 2	38	13	792	Mach-O
avionic 1	764	40	26232192	ELF
avionic 2	523	14	433472	ELF
zlib	163	40	1700	Mach-O

Table 4. Sizes of the test programs

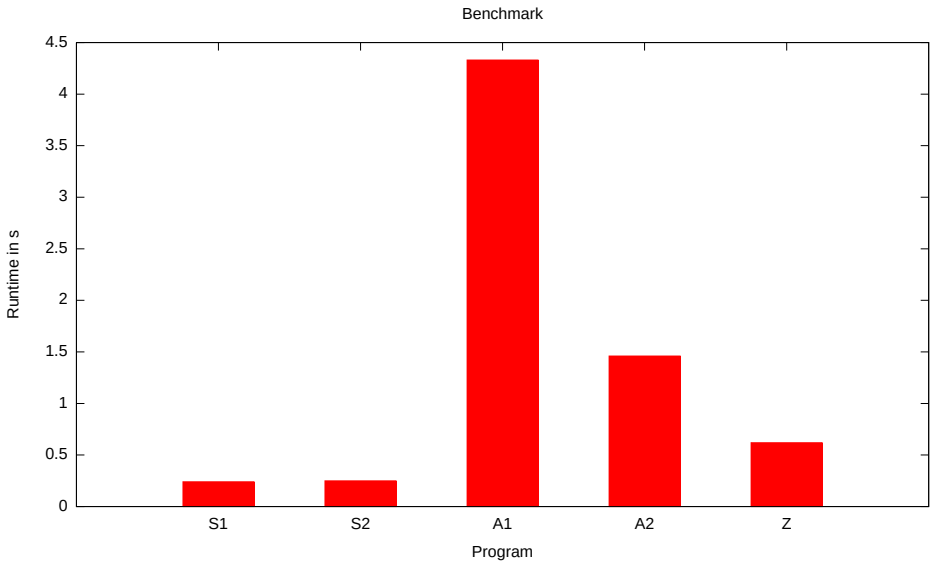


Figure 5. Runtime of the flow constraint analysis for several test programs

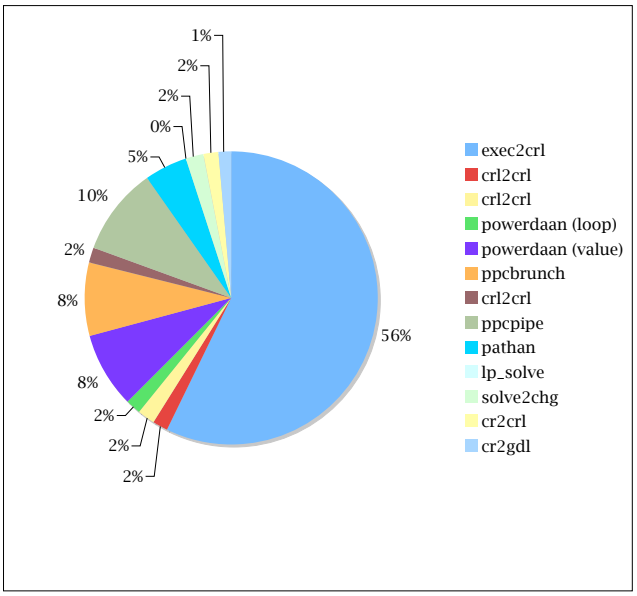


Figure 6. Overall runtime of the WCET analysis for avionic 2 broken down into subprograms

[2] Esterel Technologies. Scade suite – the standard for the development of safety-critical embedded software in the avionics industry.

[3] F. Martin. Pag - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[4] Omega Project Team. The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. 2007.

[5] M. Schlickling. Generisches slicing auf maschinencode. Master's thesis, Universität des Saarlandes, Saarbrücken, 2005.

WCET Analysis: The Annotation Language Challenge *

Raimund Kirner†, Jens Knoop‡, Adrian Prantl‡, Markus Schordan‡, Ingomar Wenzel†
Vienna University of Technology, Austria

Abstract

Worst-case execution time (WCET) analysis is indispensable for the successful design and development of systems, which, in addition to their functional constraints, have to satisfy hard real-time constraints. The expressiveness and usability of annotation languages, which are used by algorithms and tools for WCET analysis in order to separate feasible from infeasible program paths, have a crucial impact on the precision and performance of these algorithms and tools. In this paper, we thus propose to complement the WCET tool challenge*, which has recently successfully been launched, by a second closely related challenge: the* WCET annotation language challenge*. We believe that contributions towards mastering this challenge will be essential for the next major step of advancing the field of WCET analysis.*

Keywords*: Worst-case execution time (WCET) analysis, annotation languages, WCET tool challenge, WCET annotation language challenge.*

1 Motivation

The precision and performance of worst-case execution time (WCET) analysis depends crucially on the identification and separation of feasible and infeasible program paths. This information can automatically be computed by appropriate tools or manually be provided by the application programmer. In both cases some dedicated language is necessary in order to annotate this information and make it amenable to a subsequent WCET analysis. Languages used for this purpose are commonly known as *annotation languages*. Over the past 15 years, an array of conceptually quite diverse proposals of annotation languages has been presented. Many of them have been used for the implementation of a WCET tool. A comprehensive survey of WCET tools and methods has been given by Wilhelm et al. [30]. Until now, however, there has been no systematic comparison of the various approaches proposed on annotation languages for WCET analysis.

We believe that this lack is not only a hurdle for students and researchers entering the field of WCET analysis, but that it also constrains the further progress and advancement of the field. In fact, after roughly two decades of vibrant and vigorous research we consider closing this gap a major step both for consolidating the state-of-the-art and for providing a new and strong stimulus for further advancing it.

The purpose and the contributions of this paper are thus three-fold: First, to identify an array of important universally valid criteria, in which the usefulness of annotation languages for WCET analysis becomes manifest. Second, to investigate and classify a selection of prototypical representatives of annotation languages used in practice along these criteria in order to shed light on the relative strengths and limitations of the different annotation concepts. And third, most important and directly based on these findings, to extend the invitation to researchers working in this field to contribute to the challenge of designing novel and superior annotation languages, which will allow the development of even more general and powerful algorithms and tools for WCET analysis.

In addition to providing a thorough, yet smooth and survey-like introduction to annotation languages used in WCET analysis for newcomers to this field, we hope that this endeavor will in fact significantly contribute to attracting the attention of researchers who are working in this field to the challenge, which we call the *WCET annotation language challenge*: the design of novel, elegant, and easy to use powerful annotation languages.

In spite of the tremendous success the research on

*This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract P18925-N13. This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/).

†Institut für Technische Informatik, Vienna University of Technology, Austria, {raimund,ingo}@vmars.tuwien.ac.at

‡Institut für Computersprachen, Vienna University of Technology, Austria, {knoop,adrian,markus}@complang.tuwien.ac.at

WCET analysis has had in the past, and the maturity and usefulness WCET tools have already achieved and proved in practice, we believe that the major next step to further advance their power and widespread dissemination in academia and industry depends crucially on the availability of more expressive and more easily useable annotation languages, which can truly seamlessly be integrated into the tool chain of WCET analysis. We consider the identification of the most important and useful features of annotation languages, the choice of a superior mix, the development and in the long-run the standardization of a language based thereon a major challenge for researchers working in the field of WCET analysis.

In this paper we present this challenge as the *WCET annotation language challenge* to the WCET research community. As pointed out, we believe that mastering it will be the key for further advancing the field of WCET analysis. But in fact, we also believe that mastering it will be essential in order to enable the recently successfully launched *WCET tool challenge*, which has attracted the attention of many WCET tool developers [9, 28], to unfold its strength and impact in full.

2 Assessment Criteria

In this section we introduce and discuss the criteria that we use throughout this paper to assess the merits of the annotation languages and mechanisms considered. We separate these criteria into the two groups of *language design* and *usability* criteria. While the characteristics of the criteria of the first group are under control when designing the language, the characteristics of the criteria of the second group are essentially an outcome of those of the first one. Additionally to these two groups of criteria, we consider a third and orthogonal issue: the existence of a *tool* using the annotation language. This is not directly related to a specific property or feature of an annotation language. In fact, the reasons why a tool has been developed, and vice versa, why not, are many-fold. They are not necessarily related to the language at all. Nonetheless, we consider the availability of a tool an indicator of the general usefulness and usability of an annotation language. Independently of this, it is also valuable as an information on its own. It is worth noting, however, that we do not assess the quality of these tools. This is indeed beyond the scope of this paper. Readers with a deeper interest in WCET tools are invited to refer to the (forthcoming) article by Wilhelm et al. presenting a survey of WCET methods and tools [30].

2.1 Language Design

Expressiveness: We consider this to be the most important criterion of all. Intuitively, expressiveness reflects the capability of an annotation language to describe control-flow paths. Especially important is here, which types of flow information can be described and which ones cannot. An interesting level of expressiveness is *completeness*. It requires that the annotation language allows to precisely describe all feasible paths of arbitrary terminating programs. Other important issues of expressiveness are the capability of an annotation language to cope with inter-procedural program flow or selected iteration ranges of loops.

Important setscrews, which allow a language designer to control the expressiveness of an annotation language, are the means and their capabilities for dealing with *loop bounds*, with *triangle loops*, and, more generally, with *context sensitivity* and the *execution order* of statements. We consider all these throughout this paper. See Section 3.1 for further details.

Annotation placement and abstraction level: The question of placement and abstraction level of annotations has an immediate impact on the usability of any annotation language. This becomes obvious when thinking in terms of the programmer's effort to use a language.

The first design decision that has to be made is concerning the location of annotations: Shall code annotations be directly placed at the locations of the source code they describe, or shall they be provided in a separate file? None of the two options is always superior and thus consistently preferable to the other. As a rule of thumb: a) if annotated manually, it is usually more convenient to annotate the source code, b) if annotated automatically by appropriate tools, the usage of separate files often turns out to be advantageous.

The second design decision concerns the issue of annotating the source code or the object code. From a (human-centered) usability perspective, source code annotations are generally preferable. This appears to be obvious, if code annotations are manually provided. But it also holds, if flow information is automatically computed by a tool. The reason is that it is often obligatory or at least desirable to verify these annotations manually, e.g., to verify that the correct execution context has been taken into account.

Closely related is the issue of establishing a mapping between source and object code: If an object code-based annotation language is to be used to express the behavior of constructs of the original programming language it is necessary to establish some correspondence

between the two. One possibility is to define a set of constructs that can still be recognized after compilation, such as loops or procedure calls. We will call these constructs *anchors*.

Programming language: Restricting the features of the programming language that can be handled is an important option when designing an annotation language in order to control the expressiveness, precision, and efficiency of subsequent WCET analyses using the language. In effect, this means to restrict the programming language to a subset. Annotation languages, for example, can be limited to *reduceable* code. Programming languages, however, are often constrained in another way, too: by the WCET calculation methods which are compatible with the annotation language at hand. Similarly, also techniques for the automatic calculation of flow information impose often further restrictions on the programming language. Floating point operations, for example, might not be supported by an annotation language.

Besides this, it is another important issue if the annotation language supports path analysis of the object code. This is crucial because it imposes additional challenges compared to path analysis at the source code level. Different from object code, for example, source code typically uses high-level control-flow statements which allow for a simple calculation of the control-flow graph (CFG). For object code, additional annotations are necessary to reconstruct the CFG precisely.

2.2 Usability

The usability of an annotation language is possibly best reflected by the skills and the amount and the complexity of work it demands from a programmer when using it. It also refers to the knowledge that is required beyond the annotation language itself, e.g. about the WCET analysis expected to make use of it, maybe even of the implementation specifics of this technique as it might affect its performance. Similarly, this holds for the amount of work required to update a program annotation in response to an update of the program. Another issue referred to concerns the ability to cope with annotations that are automatically provided by a tool.

In principle, there are two potential classes of users that provide code annotations: a) programmers, who write manual code annotations, and b) tools that calculate annotations by means of code analysis.

Considering manual code annotations it is quite important that the program behavior can be described concisely and compactly. As an extreme case, the size of an annotation describing a specific program property, may grow exponentially with the program size. When using automatic techniques to calculate code annotations, it is important whether the techniques are capable to produce information in a format which is supported by the annotation language.

When post-processing the calculated WCET results, it is an important issue whether the WCET calculation methods compatible with the annotation language are able to provide the user with information explaining the WCET results. Standard use of ILP (cf. Section 3.4.3) with flow constraints, for example, can only provide information about the execution frequency of statements, but does not provide any information on the execution order.

The preceding discussion shows that usability is the outcome of the interplay of several factors, in particular, of the complex interplay of the annotation language and the possible support for applying this language that is provided by the (tool) environment it is used in. Assessing the usability of an annotation language thus implicitly amounts to an assessment of its usability with respect to a specific global environment, which might even change over time. This, however, is beyond the scope of this paper. For the purpose of this paper, we will thus additionally use a second term, which we call *intricacy* of an annotation language. In distinction to usability, which is the broader, the more general term, intricacy is the more specific one. We refer to intricacy to assess the language-inherent conceptual and technical complexity of an annotation language, detached from any environment or tool support of using it.

2.3 Tool Support

As mentioned above, the availability of a tool using a specific annotation language can be considered an indicator of the general usefulness and usability of this language. In particular, it is an information which we consider valuable on its own. In general, we believe that the efficiency of the known WCET calculation methods, which are compatible with an annotation language, is one of the most relevant factors driving the development of tools. It is also worth noting, however, that vice versa the efficiency of a specific WCET calculation method depends much on the specifics of the underlying annotation language. Obviously, this holds for annotation languages, which require the program structure to be unrolled in order to make the code annotation applicable. We would like to remind the reader that we are not aiming at assessing the quality of tools in this paper.

3 WCET Fundamentals

In this section we recall the essentials of flow information and of WCET calculation methods. This provides the foundation for reviewing the annotation languages we selected as prototypical representatives of the different annotation concepts. To enable this, we first divide the different kinds of flow-information into three types (Section 3.1), and then characterize the flow information which must be supported at a minimum by any reasonable annotation language (Section 3.2). The precision of flow information is limited by the expressiveness of the annotation language (Section 3.3). Subsequently, we describe the essence of the fundamental WCET calculation methods used in practice (Section 3.4).

3.1 Types of Flow Information

The static description of a program's control flow is given by its control-flow graph and its call graph. To calculate the WCET of a program also information about the dynamic control-flow behavior is needed. In WCET analysis, flow information about the dynamic control flow is typically used to partially describe the following program behavior:

Explicit execution frequency. This type of flow information describes the execution count of nodes or edges of the control-flow graph. Execution count information can be given, for example, as the absolute execution count of a code location, or as a relation between the execution count of one code location and another code location. In practice, information on execution frequency is formulated as linear equations between the execution count of different code locations.

Explicit execution order. This type of flow information is concerned with describing patterns of execution order of nodes or edges of the control-flow graph. The execution order of statements is significant on modern processors where the execution time of an instruction depends on the execution history.

Context-sensitive flow information. This refers to the control flow of instructions that may be executed multiple times within a program execution. In greater detail, we can distinguish two sources of context-sensitive flow information: instructions executed within a loop and instructions executed within a function which is called multiple times.

- *Loop-context sensitive* flow information describes the control-flow behavior of a loop body for a subrange of all possible loop iterations.
- *Call-context sensitive* flow information describes the control-flow behavior of a function for specific call sites.

3.2 Minimal Flow Information

Besides the control-flow description of a program, the only additional flow information mandatory to bound its WCET are boundaries of the execution frequency of cycles in the description of the syntactical control flow. For intra-procedural WCET analysis, the *control-flow graph* (CFG) is used as control-flow description; for inter-procedural WCET analysis the *super graph* is used, which is a combination of the call graph and the CFG of each subroutine.

In case of reducible loops [1] so-called *loop bounds* are used to describe the maximum iteration count of loops. Annotating other cyclic control-flow like recursive function calls or non-reducible loops is less intuitive.

3.3 Completeness of an Annotation Language

The *completeness* of an annotation language is concerned with the question of how precise the set of feasible control-flow paths of programs can be described by a flow annotation language. A path annotation language is *complete* if it is expressive enough to describe for arbitrary programs the *feasible paths* and the *infeasible paths* as two disjoint (non-overlapping) sets of paths, i.e., feasible and infeasible paths can be described precisely instead of having to use any approximations. For this definition of *completeness* it is sufficient to assume an implicit description of the infeasible paths by the inverse of the set of feasible paths. Given a flow annotation language which does not allow to describe the set of feasible paths precisely, one has to use over-approximations, representing a superset of the set of feasible paths. Lacking completeness in the flow description will generally result in an overestimation of the WCET.

3.4 WCET Calculation Methods

The type of interesting flow information depends much on the applied WCET calculation method. In the following we recall the three most important WCET calculation methods.

3.4.1 Timing Schema

The *timing schema* approach turned out to be an efficient WCET calculation method that is also very simple to implement [24, 26, 23]. Essentially, the *timing schema* consists of hierarchical WCET calculation rules for each node of the syntax tree representing elementary or composed statements. Denoting the local WCET bound of a node A by $T(A)$, the local WCET of the sequential composition $A;\ B$ of two nodes A and B is computed as $T(A)+T(B)$. Analogously, the local WCET of a conditional statement `if` A `then` B `else` C `fi;` is computed as $T(A)+max\,(T(B),T(C))$, while the local WCET bound of a loop `while` A `do` B `od;` with at most LB iterations (loop bound) is computed as $(LB+1)\cdot T(A)+LB\cdot T(B)$. Last but not least, if A represents an elementary statement, $T(A)$ is simply the maximum execution time of A. Of course, the *timing schema* can analogously be formulated to calculate the best-case execution time. The computational complexity of the *timing schema* is linear with the program size. It can thus be applied efficiently to large programs.

A refinement of the *timing schema* approach to handle nested loops more precisely has been presented by Colin and Puaut [6].

3.4.2 Path-Based WCET Calculation

Path-based WCET calculation [10, 27] is inspired by the naive approach of analyzing each program path and selecting the longest out of it as the WCET bound. Though this approach is infeasible for the whole program, it becomes realistic for local scopes of a program. Thus, the idea of path-based WCET calculation is to search for the longest path within each innermost scope. For example, each loop could form a scope. Once the longest path of a scope has been determined, the whole scope is treated as a single instruction with the execution time of the longest path assigned to it. This procedure is repeated till the whole program is analyzed.

Path-based WCET calculation has been developed to analyze the effects of pipelines. It allows to model the impact of the pipeline to an instruction sequence longer than just basic blocks, and thus increases the precision of the WCET bound. However, path-based WCET calculation is inappropriate to take rather global timing effects into account, like cache behavior.

3.4.3 IPET-Based WCET Calculation

The *implicit path enumeration technique* (*IPET*) has been introduced by Li and Malik [17], as well as by Puschner and Schedl [25]. In contrast to path-based WCET calculation where paths are explicitly enumerated, IPET performs an implicit longest path search.

The basic idea is to model the control flow of the program by constraints. To reduce the complexity, typically only linear constraints are used, i.e., the program is represented as an *integer linear program* (*ILP*). Subsequently to this basic modelling, supplemental flow information can be included smoothly as additional constraints of the ILP problem. The finally formulated ILP problem is passed to an ILP solver that computes the desired WCET bound. Due to the broad availability of commercial and open-source ILP solvers, such ILP problems can be solved conveniently.

4 WCET Annotation Languages

Together with Section 5 and Section 6, Section 4 represents the core of this paper. In this section we reconsider a selection of prototypical representatives of the different annotation languages (Section 4.1 to 4.7). The findings of this reconsideration will then be the basis of our conceptional comparison of these languages in Section 5.

4.1 TAL - Equations with Event Markers

Mok et al. describe the *Timing Analysis Language* (*TAL*) [20]. This is an integral part of the timing analysis system developed at the University of Texas. The timing analysis system uses the *timing schema* approach and consists of several tools retrieving information that is to be used as input for the *timetool*, which eventually performs the calculation of the execution time of the analyzed program.

While *timetool* itself works only on assembler code, the tool set also contains a modified C compiler to translate annotated C programs to annotated assembler programs. The annotations of the C code are automatically generated by the *annotate* tool that fills in default assumptions about the program's behavior. The compiler generates the annotations of the assembler code in form of a TAL script. Usually, this script is not yet useful for the analysis since it contains too conservative estimates. It has thus to be refined by a more powerful tool or by hand to get better results. To aid the user with this task, a graphical user interface is provided.

Finally, the script is interpreted by *timetool* to calculate the execution time of the program. A very detailed description of the language can be found in [5].

Figure 1 shows a simple C-program taken from [20] that will serve us as an example. The automatically

```
1 main() {                  // -v-L1:
2   int i=0, j=0; 3
3   while (i < 100) {       // -v-L3:
4     if (i < 10) j++;
5     i++;
6   }                       // -^-L6:
7 }                         // -^-L7:
```

Figure 1. Example C-Source

generated TAL-script is displayed in Figure 2. The script contains references to labels that occur in the assembler output of the compiler. We have also inserted the locations of these labels into the C-source in Figure 1.

```
1 func TAL_main() {
2   block blk1;
3   loop lp1;
4
5   blk1#begin = "-v-L1";
6   blk#end   = "-^-L7";
7
8   lp1#begin = "-v-L3";
9   lp1#count = MAXINT;
10  lp1#end   = "-^-L6";
11
12  return(blk1#time);
13 }
```

Figure 2. Autogenerated TAL-Script for the program in Figure 1

As can be seen in Figure 2, the language offers the following data types for timing purposes: A *loop* describes a loop construct where the execution frequency depends on the data being processed. A *block* is a program fragment that may contain loops, but the execution time of the block must be fixed. The language then also defines an *action*, which is any larger program fragment whose execution time is of interest. TAL distinguishes primitive and composite actions.

Each object is associated a set of attributes, such as the `time` and (loop-)`count` expressions. The syntax of assigning an attribute is `object#attribute = expression`.

In the example, there are two obvious modifications a programmer can be expected to make to the autogenerated script: First, replacing `MAXINT` as loop count attribute of `lp1` by a more accurate value.

```
9   lp1#count = 100;
```

Second, parameterizing function definition and changing the calculation formula in the last line of the script to reflect the fact that the inner `if`-statement is executed only ten times:

```
1 func TAL_main(if_count)
```

and

```
12  return (lp1#count - if_count)*blk#2time;
```

It is an interesting feature of the annotation language that it allows to specify nearly perfect execution time bounds, since the formula may contain almost any expressions. This creates new responsibilities for the programmers, who have to devise the correct calculations on their own.

4.2 Path Language and IDL

Park and Shaw proposed a WCET analysis for a subset of the C language, compiled by the GCC compiler for the MC68010 processor [26, 23, 21, 22]. They also developed the *timing schema* recalled in Section 3.4.1 for calculating the WCET of a program.

4.2.1 Path Language (PL)

Park and Shaw took much care in order to allow the specification of (in)feasible program paths. They developed a so-called *path language* (in the following called *PL*) based on regular expressions, which is shown in Figure 3. The basic idea is to label instructions interesting for path characterization with labels. Using PL one can describe path patterns, representing a set of paths.

`path` ::

a regular expression of `symbols`

`symbols` ::

alphabets(Σ) : a set of code labels

operators : $+$, $\cdot$, $\star$, $\cap$, $\neg$

parenthesis : (,)

empty set : $\emptyset$

wild cards :

$*$ . . . arbitrary string of labels: $(\Sigma)\star$

$\_$ . . . any string of code labels not containing its surrounding labels,
i.e., $x\_y = x(\Sigma - \{x, y\}) \star y$
'$\_$' may be also used as unary operator:
$\_y = (\Sigma - \{y\}) \star y$, $x\_ = x(\Sigma - \{x\})\star$

Note the difference between the Kleene star '$\star$' and the wildcard '$*$'!

Figure 3. Path language based on regular expressions [22]

Multiple occurrences of a pattern can be abbreviated, e.g., A^{2-4} is a short hand for $AA+AAA+AAAA$.

Using this convention, it can be easily expressed, for example, that a loop, where the beginning of the body is labelled LB, has an iteration count of at most 10: $\_(LB\_)^{0-10}$.

A key feature of PL is that it allows to describe patterns of explicit execution order of labeled statements. In fact, it is complete, i.e., it allows to describe all paths of terminating programs. To specify the PL expression of a given program, one has to instantiate the possible shape of input data, i.e., one has to take into account the possible valuations of input data.

A drawback of PL is that even common path patterns can result in very long expressions. For example, linear flow constraints like $f_i < f_j$ (i.e., control-flow edge f_i is executed less frequently than edge f_j) can only be described by explicitly enumerating all possible path combinations containing f_i and f_j.

Path analysis based on regular expressions can be rather computation-intensive. Every path information I_i represents a set of paths IP_i. The path analysis is done by intersecting the set of syntactically possible paths AP with the set of paths described by all the path informations $IP = \bigcap_i IP_i$. The set of feasible paths XP is then calculated as

$$XP = AP \cap IP$$

The problem with this approach is that the central path processing operations $\neg$ and $\cap$ require exponential time in general [19]. Park and Shaw thus found that PL in its generic form results in expressions too complex for path processing. Therefore, they complemented PL with a higher level *information description language*, which we are going to recall next.

4.2.2 Information Description Language (IDL)

In order to overcome the deficiencies of PL, Park et al. developed the *information description language* (*IDL*), which can be translated into a structured subset of PL [22]. For example, the information that label A and B can only be executed together is expressed in IDL as `samepath(A,B)`. This is translated to the equivalent low-level PL-expression $(*A*) \cap (*B*) + \neg(*A*) \cap \neg(*B*)$. As another example, a loop of scope A with constant iteration count K is written as `loop A K times`. This is translated into the low-level expression $\neg(*A*) + (\_A.entry\_A.body(\_A.body)^K) \star \_$. This transformation of loop information also illustrates the difficulty of getting descriptions using low-level regular expressions right. In fact, the original transformation given in [22] is faulty, as it does not take care of the case that a loop may be nested within another loop. The original translation was like $\neg(*A*) + \_A.entry\_A.body(\_A.body)^K\_$, which in case of nested loops would erroneously exclude paths with multiple executions of the loop.

The strength of IDL (as well as of PL) is that they both allow to describe path patterns of explicit execution order. However, IDL inherits a significant weakness from PL: information about relative execution frequencies of code can only be expressed by explicitly enumerating all possible path patterns, which can be of exponential length. An example illustrating this phenomenon is shown in Column 4 of Table 2 (Benchmark B2).

4.3 Linear Flow Constraints

Linear flow constraints are used in the context of IPET WCET calculation methods. The general ILP problem representing the program execution consists of n decision variables x_1, ..., x_n, an objective function $Z = \sum_{i=i}^{n} c_i \cdot x_i$ that has to be maximized, m functional constraints $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for all $i \in [1, m]$ with a_{ij} being integer constants, and the non-negativity constraints $x_i \geq 0$.

To model the WCET calculation as an ILP problem, the static program structure is reflected by the control-flow graph $G = (V, E)$, having a unique start node $s \in V$ and a unique termination node $t \in V$. The execution time of each edge $\langle i, j \rangle \in E$ is denoted by $t_{i,j}$. Denoting the execution frequency of edge $\langle i, j \rangle \in E$ as $f_{i,j}$, the WCET of a program P is given by the following objective function to be maximized:

$$wcet(P) = max \sum_{\langle i,j \rangle \in E} f_{i,j} \cdot t_{i,j}$$

The key idea to map the WCET calculation problem onto the general ILP problem is formulating the CFG structure as flow equations. For that purpose, the structure of the CFG is represented as functional constraints in the ILP problem. The CFG resulting from the source code of benchmark B_1 and B_2 (Table 2) is used as example CFG within this section. For each node exactly one flow equation is generated stating that the sum of the execution frequencies of incoming control-flow edges equals the sum of the execution frequencies of the outgoing edges. For instance, for node 5 this equation is $f_{4,5} = f_{5,6} + f_{5,8}$. To model that the program is executed exactly one time we set the frequency of the back edge $\langle 14, 1 \rangle$ to one, i.e., $f_{14,1} = 1$.

To get the WCET bound, an ILP solver is used to calculate the length of the longest possible path through the CFG. However, in the CFG the length of

this path is not bounded due to the cycle introduced by the back edge $\langle 12, 4\rangle \in E$ of the loop. Thus, it is required to add a constraint limiting the iteration count (and thus the frequency $f_{12,4}$). This is accomplished by adding an additional constraint of the form $f_{12,4} \leq LOOP\_BOUND \cdot f_{2,4}$. This so-called *loop bound* is a mandatory flow fact to calculate a WCET bound.

After this step, the obtained model can be solved by an ILP solver. There exist many implementations of such solvers, for instance the *GNU Linear Programming Kit* (*GLPK*). Figure 4 shows the resulting ILP problem of benchmark B_1 and B_2 (Table 2). The corresponding $t_{i,j}$ represent the execution times of the edges $\langle i, j\rangle \in E$ and are the coefficients of the respective $f_{i,j}$ within the objective function (in this example, for all edges $\langle i, j\rangle \in E$ it holds that $t_{i,j} = 1$).

```
Maximize

etime: 1 f1_3  + 1 f3_4 + 1 f4_5 + 1 f5_6  + 1 f5_8  +
       1 f6_9  + 1 f8_9 + 1 f9_10 + 1 f10_11 + 1 f11_12 +
       1 f10_12 + 1 f12_4 + 1 f4_13

Subject To
  f13_1  - f1_3                  = 0
  f1_3   - f3_4                  = 0
  f3_4   + f12_4  - f4_5 - f4_13 = 0
  f4_5   - f5_6   - f5_8         = 0
  f5_6   - f6_9                  = 0
  f5_8   - f8_9                  = 0
  f6_9   + f8_9   - f9_10        = 0
  f9_10  - f10_12 - f10_11       = 0
  f10_11 - f11_12                = 0
  f10_12 + f11_12 - f12_4        = 0
  f4_13  - f13_1                 = 0

  f13_1 = 1  \* Artificial back edge *\
  f12_4 - 100 f3_4 ≤ 0 \* Loop bound *\

  f5_6 - f10_11 = 0 \* Constraint B1 *\
  f5_6 - f5_8 ≤ 0 \* Constraint B2 *\
End
```

Figure 4. ILP problem for the CFG of benchmark B_1 and B_2 in Table 2.

This example illustrates that flow facts are indispensable for providing a limit on the number of the loop iterations whenever loops are present. An example for an annotation language that allows to express linear flow constraints is WCETC [15]. Another annotation language, being used within a tool that automatically extracts control-flow information and constraints from a program, has been proposed by Engblom et al. [7]. This annotation language is discussed next.

4.3.1 Modeling Contexts within Flow Constraints

The annotation language developed by Engblom and Ermedahl allows to represent flow facts over all iterations of a loop as well as over some specific iterations [7]. In particular, it also allows to specify flow facts for irreducible control flow. For WCET analysis the flow facts are converted to a format suitable for a WCET calculation method based on the *implicit path enumeration technique* (*IPET*) (cf. Section 3.4.3). Engblom and Ermedahl assume that flow-analysis is performed prior to low-level analysis, meaning that flow analysis does not have access to information about the execution time of code. The outcome of the flow analysis is a set of statically feasible paths. The WCET calculation uses information about the execution time of each piece of code to find the paths in the set of statically feasible paths that correspond to the actual worst-case execution times.

To represent the dynamic behavior of a program Engblom and Ermedahl introduce the concept of a scope. A scope has a header node that dominates all nodes in the scope and corresponds to a certain repeating execution environment, such as a function call or a loop. All scopes are supposed to be looping, even if they just iterate zero or one time. Each scope is represented by a set of nodes and edges. Scopes are connected by edges according to the control flow in the program. Every scope has a set of associated flow information facts. A flow information fact consists of three parts: i) the name of the scope, where the fact is defined, ii) a context specifier, and iii) a constraint expression. A context specifier allows to specify the iterations of the scope in which the constraint expression must be valid. The specifiers are defined using two dimensions of type and iteration space. The type allows to specify that the fact is considered a sum over all iterations, or for each single iteration separately. The iteration space is the set of iterations of the scope it is valid for. This can either be all iterations or some specified range. The flow information specified by annotations is converted to a form appropriate for IPET by mapping the scope-local semantics to execution-global semantics.

4.4 Data Value Assertions used in SPARK Ada

Chapman et al. described a WCET analysis for SPARK Ada [3, 4], the programming language used in the *Spark Proof and Timing System* (*SPATS*). The SPARK[1] language is a subset of Ada83 that is extended

[1] SPARK is an acronym for *SPADE Ada Kernel*, where SPADE is a short hand for *Southampton Program Analysis Development Environment*.

by a special kind of comments. The annotations are used for both program proof and timing analysis. Like the program proof framework, the WCET calculation in SPARK Ada is based on *symbolic execution*.

The edges of the control-flow graph of the input program are provided with weights that describe the execution time of the corresponding instructions. To keep flexibility, the weights in the CFG are given in the form of symbolic expressions instead of specific timing values; this representation has the advantage of being independent of the target hardware.

The static semantics of the SPARK language ensures that the programmer places at least one assertion before every loop statement as well as preconditions and postconditions to each function. These assertions are called *cut points*. Thus, the control-flow graph can be decomposed into a set of cut points and *basic paths* connecting them.

The problem of finding the WCET is equivalent to finding the longest path of the extended control-flow graph, which can be solved by a simplified version of the algorithm described by Tarjan [29]: through a set of transformation rules, an acyclic directed graph is mapped to a regular expression that is used to find the shortest path. The dual problem is considered in SPARK. To handle loops, a special bounded iteration operator is included in the regular expression syntax. Chapman gives three graph rewriting rules [3] to collapse alternatives, inner loops and outer loops to a simplified graph containing fewer edges, but more complex regular expression as weights. SPARK Ada expects the programmer to supply annotations for the loop bounds.

Figure 5 shows an example taken from [4] of a program calculating the power function. A distinct feature of the language is the inclusion of *modes*. SPARK Ada allows the user to specify multiple behaviors for a function that may be called from different contexts or with different input values. For each mode, the user can specify a distinct set of annotations; thereby enabling a more precise analysis.

Due to the nature of the annotations, however, it is not possible to specify tight bounds for nested loops, where the iteration space of the inner loop depends on the state of the outer loop.

4.5 Symbolic Annotations

Blieberger proposed an approach, which combines aspects of a pure annotation language with those of a programming language extension [2]. The clue of this approach is the invention of so-called *discrete loops*. Discrete loops can be considered a generalized kind of for-loops. Discrete loops allow a very flexible update of

```
1 --# proof function pow(FLOAT,INTEGER) return FLOAT;
2 function POWER(BASE: in FLOAT;
3                EXPONENT: in INTEGER) return FLOAT
4 --# pre true;
5 --# mode A (EXPONENT >=0);
6 --# mode B (EXPONENT < 0);
7 --# post (POWER = pow(BASE,EXPONENT));
8 is
9   ONE: constant FLOAT := 1.0;
10  EXCHANGE: BOOLEAN;
11  L_RES: FLOAT;
12  L_EXP: INTEGER;
13  RESULT: FLOAT;
14 begin
15  L_RES := ONE;
16  if EXPONENT ≥ 0 then
17    EXCHANGE := FALSE;
18    L_EXP := EXPONENT;
19  else
20    L_EXP := -EXPONENT;
21    EXCHANGE := TRUE;
22  end if;
23  --# loopcount(L_EXP);
24  loop
25    --# assert
26    --# ((not EXCHANGE) -> (L_RES = pow(BASE,(EXPONENT
 - L_EXP)))) in A;
27    --# & (EXCHANGE -> (L_RES = pow(BASE,(-EXPONENT -
 L_EXP)))) in B;
28    exit when L_EXP = 0;
29    L_RES := L_RES*BASE;
30    L_EXP := L_EXP-1;
31  end loop;
32  if EXCHANGE = TRUE then RESULT := ONE / L_RES;
33  else RESULT := L_RES; end if;
34  return RESULT;
35 end POWER;
```

Figure 5. An example of an annotated SPARK Ada program as given in [4]

the loop-variable, much more flexible as for a for-loop. Nonetheless, like for for-loops, also for discrete loops the loop bounds can often automatically be computed by means of reasonably simple mathematical reasoning. Particularly well-suited for this purpose are methods for symbolic analysis. We thus coin the term *symbolic annotation* for this approach here.

The following program fragment illustrates the essence underlying the concept of discrete loops:

```
1 k:= ...;
2 discrete h := k in 1..N/2
3         new h := 2*h | 2*h+1 loop
4 <loop body>
5 end loop
```

Marked by the new key word `discrete` the expression following the initialization of the loop variable `h` specifies the range both the initial value of `h` as well as all other values of `h` during subsequent iterations of the loop must be inside. Once the value is outside of this

range, the loop terminates. This captures the language extension portion of this concept. The annotation language portion is captured by the term following the keyword `new`. This term specifies the set of legal values of the loop variable of immediately adjacent loop iterations. In the example above, the new value must be either the result of doubling the old value (`2*h`), or the increment of this value (`2*h+1`). The semantics given to discrete loops requires that these constraints are validated at compile-time, or checked at run-time, if the former fails.

A very appealing feature of this approach is the seamless integration of the annotation and the program source text. This elegance, however, comes at the cost that algorithms, whose textbook version may often make deliberate use of arbitrary loops, have to be adopted or replaced by newly invented algorithms which comply with the programming discipline imposed by discrete loops. Depending on the algorithmic problem, this can be natural and easy, but sometimes also difficult and challenging, or impossible at all.

4.6 The Annotation Language of Bound-T

In [13], Holsti et al. introduce *Bound-T*, an industrially available WCET tool originally developed by Space Systems Finland Ltd and currently marketed by Tidorum Ltd. Bound-T operates on the object-code level and relies on debug information and additional assertions provided by the programmer.

The analysis is performed in three distinct phases. First, a control-flow analysis is performed to construct the call-graph of the program. The WCET calculation is then performed bottom-up on the call-graph. Bound-T cannot handle cyclic call graphs.

In the next step, iteration bounds for the loop constructs in the program are calculated. In some cases these bounds can be found by the data flow analysis that is implemented in Bound-T. In this step, the semantics of the loop body is expressed as the functional composition of the effect of the individual statements, which are expressed in Presburger arithmetic, a decidable subset of integer arithmetic. On this basis, loop increments can be found and thus can be the bounds for all counter-based loops. If Bound-T is unable to bound a loop automatically, the user is prompted to provide an *assertion* containing the loop bound. The tool will emit a warning for each instance, together with the context of the loop in question. The assertions are placed in an additional file. The decision for the external annotation is motivated by the need to support multiple execution contexts for each function. Once the call-graph has been constructed and the loop bounds have been found, the actual worst-case execution path is searched for.

On modern processors, the execution time of a particular instruction depends on the history of instructions that have previously been issued. Bound-T handles this by simulating the processor pipeline. It does not, however, model any cache behavior. The calculation is performed by transforming the analysis data into an ILP problem which is then passed to the `lp_solve` tool.

The assertion language was conceived to be flexible enough to be used with programs written in both high-level languages and assembler. Assertions are stated for a specific *scope* (= subprogram, loop or call) which are identified through their respective name or - in the case of loops - their syntactical structure. This allows for characterizations such as loops being nested inside other loops and loops calling a particular subprogram. An example of such a characterization is given in Figure 6.

```
1 loop that
2    is in (loop that calls "Foo")
3    and contains (loop that not calls "Bar"
4    and calls "Fee")
5    and not contains (loop that calls "Fee2")
6 repeats 10 times end loop
```

Figure 6. An example of a Bound-T annotation as given in [13]

While the annotations are conceptually driven by the sources, they are referentially depending on the object code. Through this design decision, Bound-T theoretically gains language and compiler independence by using the object code as a basis for its calculations, but there are also limitations that arise with this approach: Because of the optimization steps performed by the compiler, the annotations are restricted to reference only program *features* that can still be recognized after compilation. We will call these features (such as calls and loops, but not if-then-else statements) anchors. A detailed description can be found in [14]. In [12], the author of Bound-T also stated that a better mapping between source and object code is planned for a future revision of the language.

4.7 The Annotation Language of aiT

Like Bound-T, the aiT WCET tool is a commercially available tool for WCET analysis. It is developed by AbsInt Angewandte Informatik GmbH, Germany, and is available for different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555. The aiT tool reads binary files as input

programs to be analyzed. To make this more effective, the tool supports a special kind of object code annotations to reconstruct the control-flow graph from the object code [8, 11]. They allow, for example, the user to annotate the possible targets of a jump instruction in order to guide the object-code parser when reconstructing the flow graph.

aiT is not included in Table 2 since the focus of this paper is on the annotation of feasible paths in general. It is worth noting that the aiT tool also includes a value analysis to automatically calculate some flow information.

5 Discussion

In order to evaluate the annotation languages reconsidered in Section 4 we created benchmarks (Table 2) covering the criteria developed in Section 2. These benchmarks highlight the features and restrictions of the respective languages.

Each benchmark B_i consists of a source program and additional flow information that is specified informally. The first column of Table 2 describes for each benchmark the flow information to be annotated. The original source codes subject to annotation are given in the second column of Table 2, their control-flow graphs in the third column. The annotated examples for each annotation language are presented in the subsequent columns.

In the following, the most interesting results of applying the annotation languages of Table 2 to these benchmarks are presented. Table 1 provides a summary of our findings.

5.1 Expressiveness

In order to assess the expressive power of the annotation language, it is necessary to understand the calculation method that is implemented by the tool. We thus begin with an overview about the different methods and will then assess how these methods apply to the modelling of different kinds of flow information:

Calculation method. Timing schema, as implemented by TAL, were the first approach to WCET analysis and provide little more than a unified framework for the programmer to specify timing calculations with.

This approach is refined by the graph-rewriting technique used in SPARK Ada. It must be noted, however, that the expressiveness of SPARK Ada is limited, since it restricts the permitted kinds of flow information to loop-bounds.

IPET-based methods that use linear flow constraints, are widely regarded as state-of-the-art and allow a more versatile constraint-based specification of flow facts. These constraints are then used as input for an ILP solver. IPET-based tools still allow the specification of loop-bounds, which can be transformed into constraints easily.

As a unique feature, PL and IDL model execution order naturally.

Loop-bounds. The minimal information necessary to perform WCET analysis is an upper bound for every loop construct; all discussed languages support this.

Triangle-loops. The IPET-based methods (linear flow constraints and Bound-T) allow to specify inequalities as further constraints in addition to loop-bounds. With this method, so-called *triangle-loops* – these are nested loops that follow a triangular pattern in the iteration space (i, j) – can be described precisely. In the case of Bound-T, annotations of triangle-loops are only possible when they contain an anchor to identify them, such as a call. These anchors are necessary to identify program fragments in the object code.

Calling context-sensitive annotations. If loop bounds depend on input parameters, the precision will benefit from a tailored annotation for each calling context. The parametrized calculation schema of TAL supports this through a functional abstraction. While Bound-T does not expose context-sensitive information to the annotation language, it is aware of context information during the automatic computation of loop bounds.

Loop context-sensitive annotations. As described in Section 4.3.1, in the presence of caches it is often beneficial to distinguish between the first and subsequent loop iterations when formulating annotations. Currently none of the surveyed languages directly supports this feature.

Application context-sensitive annotations. SPARK Ada has a unique feature called *modes* to describe multiple annotations for a function depending on different input parameters. While this is in some ways related to calling context-sensitivity, it is not entirely the same concept. We will thus call this feature *application context sensitivity*.

Execution order. Most WCET-calculation methods content themselves with estimates of the execution

Criteria	Annotation Language							
	TAL	*PL and IDL*	*Linear Flow Constraints*	*Bound-T*	*aiT*	*SPARK Ada*	*Symbolic Annotations*	*Challenge*
Expressiveness	Timing schema	Regular expressions	Constraint-based	Constraint-based	Constraint-based	Loop-bounds	Loop-annotations	
Loop-bounds	yes	yes	yes	yes	yes	yes	yes	yes
Triangle-loops	yes	no	yes	some	yes	no	yes	yes
Calling context	yes	no	possible	implicit	no	explicit	no	yes
Loop context	no	no	possible	no	no	no	no	yes
Appl. context	no	no	no	no	yes	yes	no	yes
Execution order	no	yes	no	no	no	no	no	yes
Intricacy of Annotations	high	medium to high	medium	medium	medium	low	low to medium	as low as possible
Annot. placement	External TAL-script	Ideally inside the source code	Ideally inside the source code	External file	External file; partially inside source code	Source code comments	Integral part of the source language	– Design Decisions –
Abstraction level								
Source code	no	yes	yes	no	yes	yes	yes	
Object code	yes	no	yes	yes	yes	no	no	
Program. language								
Implementation	As'mbler/C	C	-	C, Ada	As'mbler/C	Ada	Ada	
General Scope	-	Any structured language	Any structured language	Any structured language	Any structured language	-	Any structured language	
Tool available	yes	no	yes	commercial	commercial	yes	prototype	yes

See also Table 2.

Table 1. Assessment summary

frequency of basic blocks. If the method supports the modelling of a complex hardware architecture[2], the execution order is equally important. In contrast to the PL and IDL, the IPET-based methods currently cannot be used to describe the execution order.

5.2 Annotation Placement and Abstraction Level

Placing annotations directly inside the source code is more convenient for the programmer, but may affect the readability of the program, especially in the case of library functions, which usually have many different call sites. One argument against an integration into the sources is that in an production setting, any modification of the source code may require a repeated audit.

The decision of the annotation placement is closely tied to that of the abstraction level. For obvious reasons, all surveyed tools that operate on the object code level (TAL and Bound-T) also choose to place the annotations in a separate file.

In general, the following three choices are available:

Abstraction	Placement	
Source Code:	inside	external file
Object Code:	(not practicable)	external file

Bound-T and TAL follow the approach to annotate the program at the object code level. This low-level representation gains independence from the compiler, but complicates the development phase where the source code is frequently changing. The interaction with the compiler is an important issue, as optimizations that change the control flow may invalidate the annotations.

5.3 Programming Language

If the WCET annotation is based entirely on the object code, the annotation language is theoretically independent of the original programming language. This advantage, however, is hardly exploited. For practical reasons, many of the surveyed implementations focus on subsets of the C language.

5.4 Intricacy of Annotations

As mentioned above, there is a trade-off between the expressiveness and the complexity of annotations.

[2] In complex hardware architectures the execution time of instructions depends on the execution history. Typical reasons for this behavior are instruction pipelines, instruction/data caches and processor parallelism.

TAL, for example, leaves many aspects of the WCET calculation to the user, who may specify almost arbitrary formulae within the script. While this approach would guarantee the highest precision, it also demands a high effort from the programmer, who has to devise the correct calculation schema with care. The annotation languages of aiT and Bound-T reduce the intricacy and support language constructs tailored to different kinds of flow information. At the other end of the spectrum, there is SPARK Ada, which is restricted to only loop-bounds that are annotated directly into the source code.

5.5 Availability of Tools

Most of the surveyed annotation mechanisms stem from an academic background, with aiT and Bound-T being notable exceptions; they are currently being marketed as commercial products. According to Praxis High Integrity Systems, we may still see a future release[3] of a SPARK Ada-based source code annotation language.

In closing of our discussion, the findings summarized in Table 1 illustrate that none of the annotation languages we considered uniformly outperforms its competitors, but instead have their own individual strengths and limitations. This became the more apparent, if we were to take further criteria into account, e.g., the possibility and ease of reconstructing the control-flow graph on the object-code level such that it precisely reflects its counterpart on the source-code level [16] or the consideration of application domains of annotation languages which go beyond pure WCET analysis. An approach for the latter has e.g. recently been proposed by Lisper [18]. Compared to the languages we considered in this paper, the language he proposes has a more state-oriented flavor. By its execution counters the language especially allows to express execution frequencies, similar to the linear flow constraints described above. In principle, the language could also be used to describe explicit execution orders, however, the resulting expressions will often be very complex.

6 The WCET Annotation Language Challenge

Reconsidering the annotation languages proposed and used so far for WCET analysis and opposing their key characteristics as summarized in Table 1 demonstrate that all these languages have their own specific profile of strengths and limitations. The demand for an annotation language, which combines the individual strengths of the known annotation languages, while simultaneously avoiding their limitations, is thus apparent. In Table 1 this demand is reflected by the right-most column denoted by "*Annotation Language Challenge*." It grasps the summarized strengths of the different annotation concepts. Developing a language (or an annotation concept), which enjoys this profile is the central challenge, which we derive from our investigation, and which we would like to present to the research community.

This challenge, however, is not the only challenge, which is suggested by the findings of our investigation. It is obvious that an annotation language and a methodology for computing the WCET of a program based on annotations given in this language are highly intertwined. Expressiveness delivered by an annotation language, which cannot be exploited by a WCET computation methodology, is in vain. Vice versa, the power of a WCET computation methodology cannot be evolved if the annotation language is too weak to express the needed information. This mutual dependence of annotation languages and WCET computation methodologies suggests two further challenges. Which annotation language serves a given WCET computation methodology best? And vice versa: which WCET computation methodology makes the best use of a given annotation language?

Of course, the meaning of "best" has to be made more precise to be practically useful. We argue that the underlying notion of the relation "better" has several dimensions, each of these leading to possibly different solutions. Besides parameters like ease of use, we consider the parameters of power and performance and the trade-off between the two most important.

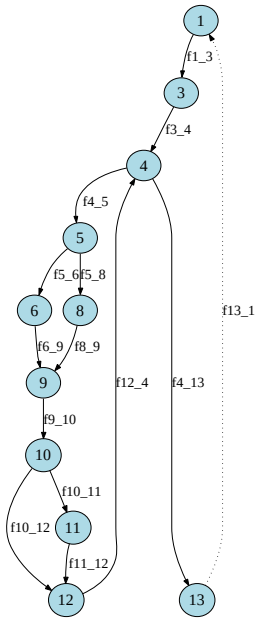
Summing up, this results in the following *challenges*:

1. Finding an annotation language, which enjoys the individual strengths of the known annotation languages while avoiding their limitations.
2. Finding an annotation language, which serves a given WCET computation methodology best.
3. Finding a WCET computation methodology, which makes the best use of a given annotation language.

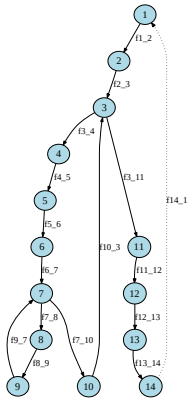
It is worth noting that these challenges can be considered on various levels of refinement, depending for instance on the notion of the relation "better" as discussed above. Thus, the challenges above represent a full array of more fine-grained challenges rather than exactly three individual challenges.

[3] http://www.praxis-his.com/sparkada/examiner.asp

1: Benchmark B_i		*2: Control flow graph*	*3: TAL*

B_1: Explicit execution order
 	
Side constraints: The conditions of the two if-statements at line 5 and 10 evaluate both to `false` or both to `true` within each iteration of the while-loop.
	1 **void** cond(**int** a[],**int** b[])
	2 {
	3 **int** i=0, j=0;
	4 **while** (i < 100) {
	5 **if** (a[i] < 10)
	6 j++;
	7 **else**
	8 a[i]=10;
	9 i++;
	10 **if** (b[i] < 10)
	11 j++;
	12 }
	13 }
			1 func TAL_cond(A_COUNT, B_COUNT) {
			2 block blk1, blk6, blk8, blk11;
			3 loop lp1;
			4 blk1#begin= "-v-LA_1";
			5 blk1#end = "-^-LA_13";
			6 lp1#begin = "-v-LA_4";
			7 lp1#count = 100;
			8 lp1#end = "-^-LA_12";
			9 blk6#begin= "-v-LA_6";
			10 blk6#end = "-^-LA_7";
			11 blk8#begin= "-v-LA_8";
			12 blk8#end = "-^-LA_9";
			13 blk11#begin = "-v-LA_11";
			14 blk11#end = "-^-LA_12";
			15 **return**(blk1#time
			16 -(min(0,100-A_COUNT)*blk6#time
			17 -(min(100,A_COUNT) *blk8#time
			18 -(min(0,100-B_COUNT)*blk11#time);
			19 }
			
			Note: The side constraints for $\mathbf{B}_1$ and $\mathbf{B}_2$ are not directly expressible in TAL; the above script represents a best effort for both $\mathbf{B}_1$ and $\mathbf{B}_2$.

B_2: Explicit execution frequency *Side constraints:* The execution frequency of the statement at line 6 is less or equal to that of the statement at line 8.			

B_3: Subranges of loop iterations
 	
In this benchmark, the challenge lies in expressing a triangular iteration pattern. (The program computes the sum $\sum_{k=1}^{n} k$)
	1 **int** compute_sum(**int** n) {
	2 **int** a=0, b=0, i=n, j, y;
	3 **while** (i>0) {
	4 a=a+1;
	5 i=i-1;
	6 j=i;
	7 **while** (j>0) {
	8 b=b+1;
	9 j=j-1;
	10 }
	11 }
	12 y=a+b;
	13 **return** y;
	14 }
			1 func TAL_compute_sum(N) {
			2 block blk1; loop lp3, lp7;
			3 blk1#begin= "-v-LA_1";
			4 blk1#end = "-^-LA_14";
			5 lp1#begin = "-v-LA_3";
			6 lp1#count = N;
			7 lp1#end = "-^-LA_11";
			8 lp2#begin = "-v-LA_7";
			9 lp2#count = N-1;
			10 lp2#end = "-^-LA_10";
			11 blk7#begin= "-v-LA_7";
			12 blk7#end = "-^-LA_10";
			13 **return**(blk1#time -
			14 N*(N-1)/2 * blk7#time);
			15 }

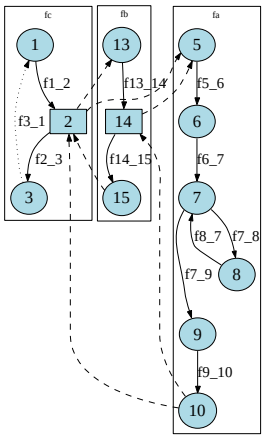
B_4: Call-context sensitive flow information
 	
Side constraints: The loop bound of the loop starting at line 7 is 10 when `fa()` is called from `fc()` and 7 when it is called from `fb()`.
	1 **int** fc(**int** m, **int** n) {
	2 **return** fa(m) + fb(n);
	3 }
	4
	5 **int** fa(**int** i) {
	6 **int** j=0;
	7 **while** (j<i) {
	8 j++;
	9 }
	10 **return** j;
	11 }
	12
	13 **int** fb(**int** i) {
	14 **return** 8 + fa(i);
	15 }
			1 func TAL_fc() {
			2 **return** TAL_fa(10) +
			3 TAL_fb(7);
			4 }
			5 /* To be called as TAL_fa(10); */
			6 func TAL_fa(I_COUNT) {
			7 block blk5; loop lp7;
			8 blk5#begin = "-v-LA_5";
			9 blk5#end = "-^-LA_11";
			10 lp7#begin = "-v-LA_7";
			11 lp7#count = I_COUNT;
			12 lp7#end = "-^-LA_9";
			13 **return**(blk5#time);
			14 }
			15 func TAL_fb(I_COUNT) {
			16 **return** TAL_fa(I_COUNT);
			17 }

Table 2: Flow Information Benchmarks and Annotation Examples, Part I

4: PL and IDL

PL:
Loop-bound:
$\neg(*L5*) + (\_L2(\_L5)^{100}) \star \_$

B_1:
$(*L6*) \cap (*L11*) + \neg(*L6*) \cap \neg(*L11*)$

B_2:
The flow relation between L6 and L8 would need full path enumeration!

IDL:
Loop-bound: `loop` $L4$ 100 `times`
B_1:
`samepath(`$L6, L11$`)`

B_2:
The flow relation between L6 and L8 is not expressible!

Note: Lx means a reference to line x of the original code

5: Linear Flow Constraints (WCETC)

Note: The side constraints for $\mathbf{B}_1$ are not directly expressible in WCETC.

B_2:

```
1 void cond (int a[], int b[]) {
2   int i=0, j=0;
3   WCET_SCOPE(s1) {
4     while (i < 100) WCET_LOOP_BOUND(100) {
5       if (a[i] < 10) {
6         j++;
7         WCET_MARKER(M1);
8       }
9       else {
10        a[i]=10;
11        WCET_MARKER(M2);
12      }
13      i++;
14      if (b[i] < 10)
15        j++;
16    }
17    WCET_RESTRICTION(M1 ≤ M2);
18  } /* scope s1 */
19 }
```

6: Bound-T annotation

B_1:

```
1 subprogram "cond"
2   loop
3     repeats 100 times;
4   end loop;
5 end "cond";
```

Note: Due to the lack of *anchor* features in the original program, a finer granularity is not possible. The side constraints for $\mathbf{B}_2$ are also not directly expressible in Bound-T.

PL:
$\neg(*L4*) + (\_L2(\_L4)^{0-n}) \star \_$
$\neg(*L8*) + (\_L6(\_L8)^{0-(n-1)}) \star \_$

$\neg(*L2\_L8*) + (\_L2(\_L8)^{\frac{n(n-1)}{2}}) \star \_$

IDL:
`loop` $L3$ n `times`
The inner loop has a variable loop bound, which is not expressible!
`execute` $L8$ $\frac{n(n-1)}{2}$ `times`
`inside` $L3$`;`

```
1 #define N 100 /* max. value of n */
2 int compute_sum(int n) {
3     int a=0, b=0, i=n, j, y;
4     WCET_SCOPE(s1) {
5       while (i>0) WCET_LOOPBOUND(N) {
6         a=a+1;
7         i=i-1;
8         j=i;
9         while (j>0) WCET_LOOPBOUND(N-1) {
10            b=b+1;
11            j=j-1;
12            WCET_MARKER(M);
13        }
14      }
15      WCET_RESTRICTION(M ≤ (N*(N-1)/2));
16    } /* scope s1 */
17    y=a+b;
18    return y;
19 }
```

```
1 subprogram "compute_sum"
2   loop that contains (loop)
3     repeats N_MAX times;
4   end loop;
5   loop that is in (loop)
6     repeats N_MAX-1 times;
7   end loop;
8 end "compute_sum";
```

Note: This assumes that `N_MAX` is a known constant

PL:
$\neg(*fb\_fa*) + (\_fb\_fa(\_L8)^{7}) \star \_$
$\neg(*fc\_fa*) + (\_fc\_fa(\_L8)^{10}) \star \_$

IDL:
`loop` $L7$ 7 `times`
`inside` fb`;`
`loop` $L7$ 10 `times`
`inside` fc`;`

```
1 int fc (int m, int n) {
2   return fa(m) + fb(n);
3 }
4 int fa (int i) {
5   int j=0;
6   /* specific loop bound for call
7   context fb(fa()) is not supported */
8   while (j<i) WCET_LOOP_BOUND(10) {
9     j++;
10  }
11  return j;
12 }
13 int fb (int i) {
14  return 8 + fa(i);
15 }
```

```
1 subprogram "fa"
2   loop
3     repeats ≤ 10 times
4   end loop;
5 end "fa";
```

Note: According to [14], Bound-T is able to perform context sensitive analysis when loop bounds depend on function parameters. It is not possible to annotate context-sensitive information directly in Bound-T.

Table 2: Flow Information Benchmarks and Annotation Examples, Part II

In order to foster research on these challenges and to assess success, we consider the reference to a collection of benchmark programs which reflect the intricacies of annotating programs for WCET analysis, and of the interaction of annotation languages and WCET computation methodologies, most valuable. Ideally, these programs should be taken from real world applications, but stripped off from unnecessary detail; focusing on just the very essence to demonstrate where current annotation languages appear insufficient or inadequate to cope with. We are planning to set up a web page to host such a library of programs. In the long run we hope that this results in a research community maintained and accepted library of benchmark programs for assessing and evaluating the relative merits of annotation languages and WCET computation methodologies and combinations thereof. In spirit this is similar to the collection of benchmark programs proposed by the organizers of the WCET Tool Challenge [9, 28]. In fact, we consider it desirable to host such libraries in close relationship to each other.

7 Conclusions and Perspectives

The power, the generality, and the ease of use of tools for WCET analysis depend strongly on the kind and the expressiveness of the annotation language used to feed the tool with program-specific WCET information. The choice of the annotation language is the most crucial decision in the early stages of designing a WCET analysis tool. This choice is not trivial. The many conflicting properties an annotation language is desired to enjoy, e.g. expressiveness vs. ease of usage and analyzability, make the choice of a "good" language indeed a challenge of its own. It is thus by no means surprising that annotation languages attracted so much attention by researchers working on WCET analysis and that so many different approaches of annotation languages have been proposed and used so far for the implementation of WCET analysis tools.

In this paper we systematically reconsidered an array of prototypical approaches which we consider pathbreaking or especially successful and important for the advancement of the new and still fast developing field of WCET analysis. The evaluation of these approaches gives indeed evidence to our thesis that the definition of a "good" annotation language is a challenge. According to our findings, which are summarized in Table 1, none of the annotation languages turns out to be uniformly superior to its competitors, let alone to be without deficiency. As discussed in Section 5, this becomes the more apparent, if further criteria are taken into account such as the possibility and ease of reconstructing the control-flow graph on the object-code level (cf. [16]) or the consideration of application domains of annotation languages beyond pure WCET analysis (cf. [18]).

In spite of the indisputably successful use of so many conceptually diverse annotation languages for WCET analysis, all this indicates that the annotation languages proposed so far are still challenged in one way or the other. It is this observation, which yields the slogan and the invitation extended by this paper:

Contributing to

> overcoming the *challenged annotation languages* by mastering the *annotation language challenge*.

We consider the invention of an annotation language, which enjoys the profile outlined in the rightmost column of Table 1 entitled "Annotation Language Challenge", as a milestone indicating the (partially) successful mastering of this challenge (and its variants). Particularly important for this will be advancements allowing a refined handling of contexts, execution orders, and interprocedural control-flow.

We believe that contributions towards mastering this new challenge will be essential for the next major step towards the further advancement of the field as a whole. The annotation language challenge complements the recently launched challenge for WCET tools [9, 28]. In fact, it is motivated by it in part. We believe that contributions towards mastering the annotation language challenge will also be a major step towards enabling the delivery of the prospects related to the tool challenge. Otherwise, the incompatibility of the annotation languages and the tools using them might soon turn out to be a significant obstacle for truly meaningful and in-depth comparisons of WCET tools.

Acknowledgments. The authors greatfully acknowledge the helpful comments of the anonymous referees and the feedback of the participants at the WCET'07 workshop. Especially, they would like to thank Niklas Holsti and Albrecht Kadlec for their many and very detailed comments, which helped to clarify and improve the presentation of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, June 1997. ISBN 0-201-10088-6.

[2] J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[3] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of spark ada. In *Proc. ACM Workshop on Language, Compiler and Tool Support for Real-time Systems*, pages K1–K11, June 1994.

[4] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[5] M. Chen. *A Timing Analysis Language - (TAL)*. Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual.

[6] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, June 2001. Technical University of Delft.

[7] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[8] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003.

[9] J. Gustafson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.

[10] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[11] R. Heckmann and C. Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.

[12] N. Holsti. Bound-t assertion language: Planned extensions. Technical report, Tidorum Ltd, 2005.

[13] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000.

[14] N. Holsti, T. Långbacka, and S. Saarinen. *Bound-T timing analysis tool User Manual*. Tidorum Ltd, 2005.

[15] R. Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[16] R. Kirner and P. Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.

[17] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[18] B. Lisper. Ideas for annotation language(s). Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen, 2005.

[19] R. MacNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(39-47), 1960.

[20] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Worksop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.

[21] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02.

[22] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[23] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.

[24] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[25] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[26] A. C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

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

[28] L. Tan and K. Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006. 13 pages.

[29] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, (Accepted January 2007).

Report from the WCET Tool Challenge 2006
Ideas for the WCET Tool Challenge 2008

Jan Gustafsson
Department of Computer Science and Electronics, Mälardalen University

jan.gustafsson@mdh.se

Abstract

The purpose of the WCET Tool Challenge is to be able to study, compare and discuss the properties of different WCET tools and approaches, to define common metrics, and to enhance the existing benchmarks. The WCET Tool Challenge has been designed to find a good balance between openness for a wide range of analysis approaches, and specific participation guidelines to provide a level playing field. This should make results transparent and facilitate friendly competition among the participants.

This short report presents conclusions from from the WCET Tool Challenge 2006 as well as some ideas for the WCET Tool Challenge 2006.

1 Goals

The goals of the WCET Tool Challenge are the following:

- To exhibit the wide range of timing analysis tools available today
  - using static program analysis, or
  - combining analysis and measurements,
  - for various target processors,
  - in various application domains,
  - supporting various programming languages and design tools,
  - academic, commercial; free or at a charge.
- To illuminate the features, abilities and intended uses of each tool
  - in finding the feasible execution paths in the SW,
  - in modelling complex processor and system HW,
  - in deriving useful WCET bounds or estimates,
  - in usability, scalability and adaptability,
  - in the range of supported targets (processors, compilers, ..)
- To collect and maintain a growing set of community standard benchmark programs and related test suites that
  - contain typical (both easy and hard) programming constructs,
  - can be analyzed by several tools with comparable results,
  - test enough of the actual behaviour of each benchmark to satisfy measurement-based tools and to validate results from static-analysis tools, and ideally, have known exact answers (paths and WCETs).

  For more details, consult the Challenge web page `http://www.idt.mdh.se/personal/jgn/challenge/`.

2 WCET Tool Challenge 2006

The first WCET Tool Challenge was performed during the autumn of 2006. It concentrated on three aspects of WCET analysis:

1. flow analysis,
2. required user interaction,
3. performance.

Two companies (developing the commercial tools aiT and Bound-T) and three research groups (developing the research tools SWEET, Chronos and MTime) participated. The actual work with the tools was made by an external user and the development teams. The evaluation was targeted on a set of benchmark programs.

The report was presented at ISoLA 2006 as two separate papers. The first [1] was the main report; the other [3] presented the experiences from the external user. There is also a technical report available [2], which contains more details.

2.1 Results from WCET Tool Challenge 2006

This first Challenge was performed successfully in spite of some initial unclearness, debate and delay. We noted that there is no common format of the results from the different tests, and that different developers have made different tests sometimes. We draw the following conclusions:

- The tests have been a real challenge to the participating WCET tools. We have a success range from 0% to 100% in terms of how many of the benchmark programs that were analyzable by a certain tool.
- The tests have clearly pointed out problems existing in the tools as well as in the benchmarks and the used compilers.
- Most of the tools find more than half of the loop bounds automatically. Only one tool finds infeasible paths automatically.
- Several bugs in both the tools and the benchmarks have been corrected during the Challenge.
- Actual WCET estimates cannot be compared this time since the developers support different processors and compilers.
- The quality of WCET estimates is hard to judge for all tools but aiT, since aiT was the only tool to provide measurements for some of the benchmarks. Chronos provided simulated values that indicate the possible size of overestimation.

3 WCET Tool Challenge 2008

Since the work to perform the Challenge is extensive, both for the work group and the developers, we decided to make the event bi-annual. Another reason was to have some tool development time between tests, so real progress could be observed from one Challenge to the next. This means that the next Challenge is to be performed 2008.

A number of actions have to be made to enhance the next Challenge compared to the first. Some of them are:

- Update benchmarks and the setup of the Challenge according to the feedback from the WCET Tool Challenge 2006.
- Extend the active working group to 2 or 3 persons.
- Create a reference group which will act as a support to the working group.
- Try to extend the number of participants. Tools like OTAWA and Heptane, and measurement-based tools (e.g., Rapita, SYMTA/P) should be invited again. There are also some new tools entering the WCET analysis scene (like TimeBounder from Korea).
- The procedure of identifying test data (inputs) necessary for measurement-based based tools should be defined.
- The inputs for the worst-case behavior of the benchmark programs should be defined.
- Be more detailed concerning tests and format of result reports.
- Exclude benchmark programs that are redundant in terms of what is tested.
- Include industrial code, e.g. aero space and automotive code (e.g., Daimler-Chrysler code).
- If possible, include more automatically generated code.
- Try to focus on one common processor (ARM7?) to be able to compare WCET estimates.
- Define hardware setup in detail so comparative low-level analysis (e.g., cache and branch prediction analysis) can be performed.

References

[1] J. Gustafsson. The worst case execution time tool challenge 2006. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Nov. 2006.

[2] J. Gustafsson. WCET challenge 2006 technical report. MRTC report 1209, 2007. Technical Report MRTC report 1209, 2007, Mälardalen University Real-Time Research Centre, Mälardalen University University, Västerås, Sweden, 2007.

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