THESIS FOR THE DEGREE OF LICENCIATE OF ENGINEERING

A Static Timing Analysis Method for Programs on High-Performance Processors

THOMAS LUNDQVIST

Department of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 1999

A Static Timing Analysis Method for Programs on High-Performance Processors

THOMAS LUNDQVIST

Technical Report no. 315L

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:

Thomas Lundqvist
Department of Computer Engineering
Chalmers University of Technology
Hörsalsvägen 11
SE–412 96 Göteborg, Sweden

Phone: +46 (0)31–772 1165
Fax: +46 (0)31–772 3663
Email: thomasl@ce.chalmers.se
URI: http://www.ce.chalmers.se/~thomasl

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 1999

A Static Timing Analysis Method for Programs on High-Performance Processors

THOMAS LUNDQVIST
Department of Computer Engineering, Chalmers University of Technology

Thesis for the degree of Licenciate of Engineering, a Swedish degree between M.Sc. and Ph.D.

Abstract

When constructing high-performance real-time systems, safe and tight estimations of the worst case execution time (WCET) of programs run on pipelined processors with caches are needed. To obtain tight estimations both path and timing analyses need to be done. Path analysis is responsible for eliminating infeasible paths in the program and timing analysis is responsible for accurately modeling the timing behavior of dynamically scheduled processors employing pipelining and caching.

This thesis presents a new method, based on cycle-level symbolic execution, that combines path and timing analyses for programs on high-performance processors. An implementation of the method has been used to estimate the WCET for a suite of programs running on a high-performance processor. The results show that by using a combined analysis, the overestimation is significantly reduced compared to previously published methods. The method automatically eliminates infeasible paths and derives path information such as loop bounds, and performs accurate timing analysis for a multiple-issue processor with an instruction and data cache. The thesis also identifies timing anomalies in dynamically scheduled processors. These anomalies invalidate the basic assumption made in previously published timing analysis methods: that the maximum instruction execution time represents the worst-case behavior. To handle these anomalies, a method based on program modifications is suggested that makes it possible to safely use all previously published timing analysis methods even for architectures where timing anomalies can occur. Finally, the use of data caching in real-time systems is examined. For data caching to be fruitful, data accesses must be predictable when estimating the WCET. The results from an empirical study show that many accesses are indeed predictable. Thus, data caching can be expected to be efficient in real-time systems.

Keywords: Real-time systems, worst-case execution time, timing analysis, path analysis, symbolic execution, multiple-issue, pipeline, caches, timing anomaly, dynamically scheduled processor.

List of Appended Papers

This thesis is a summary of the following three papers. References to the papers will be made using the Roman numbers associated with the papers.

I. Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. To appear in *Special Issue on Timing Validation, Journal of Real-Time Systems*, November 1999.

II. Thomas Lundqvist and Per Stenström. *Timing Anomalies in Dynamically Scheduled Microprocessors*. Technical Report 99–5, Department of Computer Engineering, Chalmers University of Technology, Sweden, April 1999. Submitted for publication.

III. Thomas Lundqvist and Per Stenström. *Empirical Bounds on Data Caching in High-Performance Real-Time Systems*. Technical Report 99–4, Department of Computer Engineering, Chalmers University of Technology, Sweden, April 1999.

1 Introduction

When constructing highly dependable real-time systems, it is often of utmost importance to certify that a task or program finishes its execution before a given deadline. This assures that a set of programs can be feasibly scheduled to run together or that a given timing constraint in the system is fulfilled. In this process, methods for estimation of the worst case execution time (WCET) of programs are needed. These methods must produce *safe* estimates—the WCET must not be underestimated—and preferably also *tight* ones—the overestimation of the WCET should be small to utilize the processor efficiently.

The WCET of a program is defined as the maximum execution time given that all or some input data and initial system state are considered to be unknown and given that the program is run uninterrupted on one processor. The input data controls which path in the program is executed. Each path then executes in a time determined by the computer system running the program. This means that the WCET corresponds to the execution time of the longest path found in the program.

In order to obtain tight WCET estimations, both path and timing analyses are required. *Path analysis* is responsible for eliminating infeasible (non-executable) paths in the program. These paths can never be executed regardless of input data and should not contribute to the WCET estimation. *Timing analysis* is responsible for giving a safe and tight estimate of the maximum execution time of a given path in the program and must account for the different features found in a computer system. Typical features are the processor and the memory system, which provides the processor with instructions and data.

High-performance general-purpose processors execute instructions in parallel using pipelined execution and dynamic scheduling of instructions. They also include instruction and data cache memories to increase the average performance. These features complicate the timing analysis and have been the focus of recent research [4–9, 11, 12, 15, 16]. Although the path analysis can be performed manually by inserting annotations in the program [13, 14], this is error-prone and time consuming and an automatic path analysis is more attractive [1–3]. No method so far has managed to integrate automatic path analysis and detailed timing analysis into the same method.

This thesis focuses on static path and timing analyses methods for high-performance general-purpose processors. Static means that the analysis is carried out before the program is executed and does not rely on run-time information. The thesis is a summary of three papers, referred to by the Roman numerals **I**–**III**. The first contribution is a new timing analysis method that integrates automatic path and timing analyses focusing on pipelined processors with instruction and data caches (**I**). Results obtained from using this method show that tight WCET estimations are obtained by eliminating infeasible paths and carrying out accurate timing analysis. The next contribution (**II**) identifies timing anomalies in dynamically scheduled processors. A timing anomaly is a counter-intuitive situation which previously published timing analysis methods [4, 6, 8, 9, 12,

15] are unable to handle. To handle these anomalies, a new method based on program modifications is presented which allows the method presented in **I** to be used also for dynamically scheduled processors. The last contribution (**III**) is a characterization of data memory accesses with respect to predictability and an empirical study of data caching showing that the majority of data accesses are predictable.

The rest of this thesis is organized as follows. The problem formulations and contributions made in **I**, **II**, and **III**, are summarized in Sections 2, 3, and 4, respectively. Section 5 finally discusses the results and points out future work.

2 A New Path and Timing Analysis Method

Previously published WCET estimation methods [4, 6, 8, 9, 12, 15] often suffer from overestimation due to inadequate path analysis. Also, they rely on the user to make manual annotations in the program to supply path information. Although some of the previously published automatic path analysis methods [1, 2] include timing analysis as well, the timing models used in these methods assign fixed execution times to each instruction or basic code block and it is not obvious how to extend these methods to handle more complex architectures with pipelined processors and caches. More seriously, no one has shown how to integrate automatic path analysis with detailed timing analysis.

In **I**, a new WCET estimation method is presented and evaluated that integrates automatic path and timing analyses. The approach is to use architectural simulation techniques [10, 17] that simulate each path in the program on cycle-level timing models of the computer system. In principle, this makes it possible for the timing analysis to be arbitrarily accurate. However, in order to estimate the WCET, all paths in the program must be analyzed. This was accomplished by extending the simulation techniques to also handle unknown data so that the program can be symbolically executed. Using this extended simulation technique, all paths in the program can be analyzed: Statically known information such as loop bounds can be derived from arbitrarily complex expressions and many infeasible paths can be eliminated.

In general, it is not feasible to use the basic approach described above as the number of paths to simulate can grow exponentially with the length of the program. To handle this, a path merge strategy was employed: Each time two simulated paths meet in the program they are merged into one. For example, in a loop containing two possible paths, these paths will be merged into one before a new iteration starts, which makes an exponential growth of the number of paths impossible.

A critical problem when merging two paths is to decide which of them corresponds to the worst case. If the state of the timing model used differs in the two paths, it is necessary to know which one to choose in order to obtain a safe WCET estimation. This problem is treated in the paper and the merge operation is extended to handle a timing model of a multiple-issue pipelined processor with instruction and data cache memories.

To evaluate the method, it was used to estimate the WCET of seven benchmark programs. First, the path analysis capability was evaluated by using a timing model of a system containing no caches and where all instructions execute in a single cycle. The results showed that the path analysis successfully eliminates many infeasible paths and derives loop bounds automatically. In all but one of the programs, the method produced an exact estimate of the WCET. Next, the timing analysis capability was evaluated by using a timing model of a dual-issue pipelined processor with instruction and data caches. In this case, an exact estimate of all but two of the programs was obtained. This shows that the merge operation performs well, even when using a more complicated timing model. Part of the overestimation in the two programs was caused by unpredictable data accesses. This topic will be further explored in Section 4. Finally, the importance of doing both path and timing analyses was examined by comparing the results with a naive method using no path analysis, and no pipeline or cache analysis. This resulted in the WCET being overestimated by a factor of 20.

3 Timing Anomalies in High-Performance Processors

In dynamically scheduled processors, the order of execution between instructions is determined dynamically at run time. The resulting schedule depends on the instruction execution time, referred to as *latency*, of each individual instruction. If the latency of an instruction changes, the future scheduling of instructions can also change.

All previously presented timing analysis methods [4, 6, 8, 9, 12, 15], including the one presented in **I**, run into a general problem if one tries to extend them to handle dynamically scheduled processors. During estimation, the latency of some instructions can be unknown. In these situations, all previous methods rely on the assumption that the maximum latency of an instruction is always the worst case latency. For example, if the outcome of a cache access is unknown, a cache miss is assumed.

In **II**, it is shown that this assumption is not always true for dynamically scheduled processors. It is possible that a shorter latency will change the scheduling of instructions so that the overall execution time is increased. Choosing the long latency can thus lead to an underestimation of the WCET. In the paper, several examples of such timing anomalies are presented. Furthermore, the architectural features that make these anomalies possible are identified as well as a necessary condition for when they can show up. It is found that if all resources used by the instructions are allocated in program order then no timing anomalies can occur.

A straightforward approach to find the WCET in the presence of timing anomalies would be to do an exhaustive search of all possible latencies and their corresponding future schedules. However, if there are n instructions with unknown latencies along a path in the program and each instruction may lead to k different future schedules, then, in the worst case, k^n different schedules must be analyzed. This is in general not feasible and **II** presents two other methods that make it possible to use previously published methods for safe estimation of the WCET: the pessimistic serial-execution

method, and the program modification method.

The pessimistic serial-execution method assumes a timing model where handling of cache misses and execution of instructions never overlap, and where instructions execute one at a time in the pipeline. The estimated WCET obtained using this method will be safe but very pessimistic. The other method, based on making program modifications, first identifies all places in the program where the timing analysis method needs to make safe local decisions regarding the outcome of an unknown event. For example, whenever a variable-latency instruction with unknown latency is found, a decision must be made regarding which latency is the worst-case one. At these places, the program is modified in order to assure a predictable state of the timing model and a safe decision. For example, variable-latency instructions can be forced to execute in-order. Then, the maximum latency will be the worst case latency.

The amount of pessimism incurred by the two methods were evaluated using seven benchmark programs. To estimate the WCET of the program, the timing analysis method presented in **I** were used together with an analytical model of a dynamically scheduled processor with instruction and data caches. The results indicate that using program modifications, safe estimations were possible without adding to much pessimism; less than 27 % of the WCET was added for the benchmark programs. The program modification method also performed well compared to the serial execution method which overestimated the WCET with at least a factor of 2. The conclusion is that for the programs studied it is possible to obtain safe and fairly tight WCET estimation even when using dynamically scheduled processors where timing anomalies may show up.

4 An Empirical Study of Data Caching

The last paper, **III**, treats the problem of using data caching in time-critical real-time systems. Data caches are critical to get high average performance in high-performance processors. However, in order for a data cache to be useful in a real-time system, data accesses originating from a program must be statically analyzable when estimating the WCET of the program. Data accesses that cannot be analyzed lead to an overestimation of the WCET. If this overestimation is too severe, it can be better to turn off data caching altogether which would lead to a big drop in performance.

In **III**, a characterization of data access instructions is made as being either predictable or unpredictable. Unpredictable memory access instructions use a reference address that depends on input data which is unknown during estimation and can therefore not be analyzed. Also, the term *unpredictable data structure* is defined as being a data structure that is accessed by at least one such unpredictable memory access instruction.

A previously presented method for handling data accesses [5] charges two cache misses to each access that the method identifies as being unpredictable. The reason for this is to account for the possibility of an additional future cache miss when replacing a

useful block. This strategy is obviously very pessimistic and in **III**, another approach is suggested. By using currently available hardware facilities it is possible to choose what to cache or not. By not caching unpredictable accesses, only one cache miss needs to be accounted for.

To answer the question of how effective data caching can be in a real-time system, a classification of data memory accesses were made by running two programs, *compress* and *swim*, from the SPEC95 benchmark suite and collecting all data accesses. These accesses were then used to classify data structures as predictable or unpredictable. It was found that 84 % of all memory accesses in *compress* and *all* memory accesses in *swim* went to predictable data structures. Thus, based on these programs, data caching is found to be effective, although, good timing analysis methods are needed to fully exploit the potential of data caching.

5 Discussion and Future Work

The timing analysis method presented in this thesis combines automatic path analysis with detailed timing analysis in order to reduce the WCET overestimations. However, this integration of path and timing analyses is not the only advantage of this method. Being based on a simulation approach, it has the potential of performing very accurate timing analysis since all iterations in a loop are analyzed (the loop is unrolled). As an important example, this can be useful when analyzing accesses going to the data cache. Consider for example the array accesses found in the *swim* application in **III**. These accesses appear in the form of a stride where the reference address used is a linear combination of one or two loop index variables. These accesses are perfectly analyzable using the basic symbolic execution method without the need for a separate array analysis as used in [5, 16].

However, since all loop iterations are analyzed, the analysis complexity of the symbolic execution method is higher than that of compiler-based methods that analyze the loop body once regardless of the number of iterations. The time to perform an analysis with the symbolic execution method will at least be proportional to the actual execution time of the analyzed program. Often, several paths need to be analyzed and even if the merge approach makes an exponential complexity impossible, the method can still experience polynomial time complexity.

This thesis treats only a small part of a computer system: statically or dynamically scheduled, pipelined processors with instruction and data caches. A real system contains additional features and mechanisms that has not been studied here. For example, branch prediction mechanisms and branch history buffers, load/store buffers, and the contention between different operations on the processor-memory bus. Therefore, further research is needed to really be able to obtain safe estimates of the WCET for programs on complete systems using high-performance processors.

Predictable data caching is also an issue that requires further study. If a program only uses unpredictable data structures, the estimated WCET becomes very pessimistic.

Other means of handling these data structures should be explored using both software and hardware methods.

The basic method presented in this thesis can also be extended when regarding path analysis. Further experimental evaluation is needed regarding the benefits of using a more powerful domain of values during simulation. A more expressive domain would make it possible to directly express knowledge about input data and to automatically identify some mutually exclusive paths in the program. Also, further design alternatives of merge operations need to be studied. It is sometimes wise to restrict the merging since information needed for eliminating infeasible paths can get destroyed by the merge operation.

Acknowledgments

I am deeply grateful to my advisor Professor Per Stenström for supporting and guiding me in my research struggles. I believe that his continuous strive for good quality research and his ability to organize and focus on problems and ideas have made this thesis into something people will read and use, instead of simply using it as yet another bookshelf decoration.

I am also grateful to all the people around me at the department, especially: Magnus Karlsson, Jim Nilsson, Jan Jonsson, Jonas Vasell, Fredrik Dahlgren, and Björn Andersson for fruitful critique and discussions regarding WCET issues. My thanks also goes to Fredrik Lundholm for keeping my computer running, and Gerd Georgsson, Madeleine Persson, and Ewa Cederheim-Wäingelin for keeping everything else running. There are also many other people that have raised the quality of this thesis by keeping me happy and comfortable. They include all people in our department but also former colleagues like Ulf Hansson, Susanne Bolin, and Gandong Li. Also, my special thanks to Holger Broman who always makes sure I have an ergonomic working position.

My wonderful wife Ulrika deserves a chapter of her own but I am a bit lazy. Thank you for everything!

This research is supported by a grant from the Swedish Research Council on Engineering Science (TFR) under contract number 221-96-214.

References

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

[2] Roderick Chapman, Alan Burns, and Andy Wellings. Integrated program proof and worst-case timing analysis of SPARK ada. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages K1–K11, June 1994.

[3] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, pages 1298–1307, August 1997.

[4] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[5] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996.

[6] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.

[7] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[8] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[9] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 334–345, December 1998.

[10] P Magnusson, F Dahlgren, H. Grahn, M. Karlsson, F. Larsson, A. Moestedt, J. Nilsson, P Stenström, and B. Werner. Simics/sun4m: A virtual workstation. In *Proceedings of USENIX98*, pages 119–130, 1998.

[11] Frank Mueller. Timing predictions for multi-level caches. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1997.

[12] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 47–55, June 1997.

[13] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5:31–62, 1993.

[14] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, 1989.

[15] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, December 1998.

[16] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3nd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.

[17] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS '96*, pages 68–79, 1996.

Paper I

An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution.

To appear in

Special Issue on Timing Validation,
Journal of Real-Time Systems,
November 1999.

To appear in Special issue on Timing Validation, Journal of Real-Time Systems, 1999
© 1999 Kluwer Academic Publishers, Boston.

An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution

THOMAS LUNDQVIST AND PER STENSTRÖM {thomasl,pers}@ce.chalmers.se

Department of Computer Engineering, Chalmers University of Technology, SE–412 96 Göteborg, Sweden

Abstract. Previously published methods for estimation of the worst-case execution time on high-performance processors with complex pipelines and multi-level memory hierarchies result in overestimations owing to insufficient path and/or timing analysis. This does not only give rise to poor utilization of processing resources but also reduces the schedulability in real-time systems.

This paper presents a method that integrates path and timing analysis to accurately predict the worst-case execution time for real-time programs on high-performance processors. The unique feature of the method is that it extends cycle-level architectural simulation techniques to enable symbolic execution with unknown input data values; it uses alternative instruction semantics to handle unknown operands.

We show that the method can exclude many infeasible (or non-executable) program paths and can calculate path information, such as bounds on number of loop iterations, without the need for manual annotations of programs. Moreover, the method is shown to accurately analyze timing properties of complex features in high-performance processors using multiple-issue pipelines and instruction and data caches. The combined path and timing analysis capability is shown to derive **exact** estimates of the worst-case execution time for six out of seven programs in our benchmark suite.

Keywords: Real-time systems, worst-case execution time, timing analysis, path analysis, symbolic execution, multiple-issue processor, caches, architecture simulation.

1. Introduction

Static estimation of the worst-case execution time (WCET) has been identified as an important problem in the design of systems for time-critical applications. One reason is that most task scheduling techniques assume that such estimations are known before run-time to schedule the application tasks so that imposed timing constraints, e.g. deadlines, are met at run-time. Another reason is to early in the design process determine the processing capacity needed to meet timing constraints of a real-time application.

The actual WCET of a program is defined by the program path that takes the longest time to execute for the entire set of possible input data and initial system states. Ideally, a WCET estimation method should take as input a program and estimate a tight upper-bound on the actual WCET for a given hardware platform. There are two important sources to overestimations of the WCET. First, the estimation technique may include program paths that can never be executed regardless of the input data, usually referred to as *infeasible,* or non-executable, program paths. Second, the timing model of the hardware platform may introduce overestimations of the WCET because of simplifying timing assumptions. Both

of these sources to overestimations result in poor resource utilization at run-time which can considerably increase the system cost and/or reduce the schedulability of real-time programs.

The first problem can be addressed by requiring that the programmer provides path annotations (Puschner and Koza, 1989) which clearly requires a considerable programming effort and is error-prone. A more attractive method is to automatically detect infeasible paths through static *path analysis methods* (Chapman et al., 1994; Ermedahl and Gustafsson, 1997; Altenbernd, 1996). As for the second problem, several *timing analysis* approaches have been proposed that statically estimate the execution time of a given path by taking into account the effects of, e.g., pipeline stalls and cache misses (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Kim et al., 1996; Lim et al., 1998; Healy et al., 1995; White et al., 1997; Theiling and Ferdinand, 1998). Unfortunately, none of the methods have managed to successfully integrate timing analysis for a high-performance processor with accurate path analysis to identify and exclude infeasible paths from the analysis automatically.

In this paper we present a new approach to static estimation of the WCET that integrates path analysis with accurate timing analysis. Our approach is to use architectural simulation techniques (Magnusson et al., 1998; Pai et al., 1997) that simulate each path on cycle-level timing models of the hardware platform. We extend such techniques to enable symbolic execution of programs in absence of knowledge of input data by augmentation of the instruction semantics to handle also unknown input data values. This results in the automatic derivation of statically known path information such as loop bounds from arbitrarily complex expressions and exclusion of many infeasible program paths.

A practical limitation of the method in its basic form is that the number of paths to simulate can easily become prohibitive, especially in loop constructs. We have therefore extended the symbolic execution method with a path-merging approach that manages to bound the number of simulated paths in a loop to just a few paths. This merging strategy makes the approach useful for realistically sized programs. We demonstrate how the method can be applied to multiple-issue pipelined processors with instruction and data caches by presenting how the merging strategy is implemented to encounter the worst-case effects of these features. We have implemented the method and used it to estimate the WCET of programs run on a simulated dual-issue pipelined processor with instruction and data caches. We have found that the implementation of the method is capable of deriving **exact** estimates of the WCET for six out of seven programs in our benchmark suite.

The next three sections are devoted to describing our method focusing on the basic symbolic execution approach in Section 2, the WCET algorithm in Section 3 and how detailed timing models of architectural features are integrated into the method in Section 4. An evaluation of the method in the context of a case study of a dual-issue processor with caches and WCET estimation of seven programs is presented in Section 5. In Section 6, we discuss the properties of the method beyond the limitations of our particular implementation. Finally, we relate our contribution to the work by others in Section 7 before we conclude in Section 8.

2. Cycle-Level Symbolic Execution

Consider programs for which the WCET is statically decidable; i.e., all possible execution paths in the program are finite in length regardless of input data. For example, bounds on number of iterations in loops are known before run-time, although they might be difficult to determine using existing (compiler-based) analysis methods. Second, for the time being, we will consider processors with fixed instruction execution times although our method is applicable to more complex processor architectures with associated memory hierarchies which we consider in Section 4. Given these assumptions, WCET could be conceptually determined by identifying the feasible path through the program with the longest execution time.

Cycle-level architectural simulation techniques have now matured so that the execution time of a program run on complex processor architectures can be accurately determined with a reasonable simulation efficiency (Magnusson et al., 1998; Pai et al., 1997). The advantages of using architectural simulation techniques are twofold. First, it is possible to make arbitrarily accurate estimations of the execution time of a program for a given set of input data. Second, and presumably more importantly, when a given path through the program is simulated, all static (i.e., input data independent) information about this path is automatically extracted.

A cycle-level architectural simulator can be seen as an instruction-level simulator connected to a clock-cycle accurate architectural timing model. In this section, we concentrate on instruction-level simulators connected to a timing model that assumes fixed instruction execution-times. First, we present how it can be extended to estimate WCET in Section 2.1. In Section 2.2 we show how the path analysis can be automated using this approach. Finally, in Section 2.3 we discuss implementation issues of the basic approach and how they are addressed.

2.1. The approach

Instruction-level simulation techniques assume that input data is known and therefore only analyze a single path through the program associated with this input data. To find the WCET of a program using this approach, however, the program would have to be run with all possible combinations of input data which is clearly not feasible. Our approach is instead to symbolically execute the program, which conceptually means that all paths through the program are simulated and in this process infeasible paths are excluded. To do this, we have extended traditional instruction-level simulation techniques with the capability to handle unknown data and with an extended semantics for each data-manipulating instruction to correctly perform arithmetics with the unknown data values as follows.

Each data type is extended with an element denoted *unknown*. In general, the semantics of all arithmetics and logical operations must be redefined to correctly calculate the result if any of the source operands have an *unknown* value. Examples of the extended semantics for some common instruction types can be seen in Table 1. Consider for example an add instruction, ADD T, A, B, that operates on 32-bit unsigned integers, $Z := \{0 \ldots 2^{32} \Leftrightarrow 1\}$ and is defined as $T \leftarrow A + B$. In the extended

semantics, it is instead defined on $\overline{Z} := Z \cup \{unknown\}$ with the semantics seen in Table 1.

Table 1. Extended semantics of instructions.

Instruction type	Example	Semantics
ALU	ADD T,A,B	$T \leftarrow \begin{cases} unknown & \text{if } A = unknown \text{ or } B = unknown \\ A + B & \text{otherwise} \end{cases}$
	AND T,A,B	$T \leftarrow \begin{cases} unknown & \text{if } A = unknown \text{ and } B \neq 0 \\ & \quad \text{or } B = unknown \text{ and } A \neq 0 \\ 0 & \text{if } A = 0 \text{ or } B = 0 \\ A \text{ and } B & \text{otherwise} \end{cases}$
Compare	CMP A,B	$A - B$ and update the condition code (cc) register. May set bits in the cc-register to *unknown*.
Conditional branch	BEQ L1	Test bit in cc-register to determine whether a branch is taken. If bit is *unknown* simulate both paths.
Load	LD R,A	Copy data from memory at address A to register R (the data can be *unknown*). If address is *unknown*, set R to *unknown*.
Store	ST R,A	Copy data from register R to memory at address A (the data can be *unknown*). If address is *unknown*, all memory locations are assigned *unknown*(A more efficient solution is discussed in Section 2.3.)

The load and stores need special treatment, since the reference address used may be *unknown*. For loads, this results in an *unknown* value being loaded into a register. For stores, however, an unknown address can modify an arbitrary memory location. Therefore, the correct action would be to assign the value *unknown* to all memory locations to capture the worst-case situation. This is of course a major limitation. For the time being, we will assume that all addresses used by store instructions are statically known. However, we will discuss efficient solutions to overcome this limitation later in Section 2.3.

The semantics for a conditional branch is also special. When a conditional branch whose branch condition is *unknown* is encountered, both paths must be simulated. On the other hand, when the branch condition is known, the extended simulation technique will exclude paths that can never be taken. Let's review this path analysis capability in some more detail in the next section.

2.2. Path analysis

To understand how the cycle-level symbolic execution technique can automate path analysis, consider the program in Figure 1 which calculates the sum of all the values in the upper-right triangle of matrix b. For simplicity, we reason about the program

```
  int fun(int b[4][4])
  {
1   int i, j, sum = 0;
2
3   for (i = 0 ; i < 4 ; i++)
4     if (b[i][i] > 0)
5       for (j = i+1 ; j < 4 ; j++)
6         if (m[i][j])
7           sum += b[i][j];
8   return sum;
  }
```

Figure 1. Example program.

in a high-level language, even if the symbolic execution of course is done at the instruction level.

In the beginning of the symbolic execution, data values in matrix b are treated as unknown input and all elements are assigned the value *unknown*. The boolean values in matrix m are considered known. When analyzing this program, the conditional branch on line 4 will be the only branch depending on unknown values. Consequently, the two possible execution paths originating from this conditional branch have to be simulated. One path continues through lines 3,4, the other one through lines 5,6. . . 3,4, until they hit the same unknown conditional branch again during the second iteration of the outer loop. Continuing this way, $2^4 = 16$ paths will reach the end of the program. WCET is the longest of these paths. The infeasible paths originating from the conditional branch at line 6 are automatically eliminated since the branch condition is determined during simulation. Also, no loop bound annotation for the inner loop is needed since the iteration count only depends on variable i which is also determined during the simulation.

An important advantage of the symbolic execution approach is that all conditions that depend on data values that are known statically will be computed during the simulation. For example, input data independent bounds on the number of loop iterations that are expressed with arbitrarily complex functions are computed automatically. This is not handled by known compiler-based analysis methods, such as used by (Healy et al., 1998). Interestingly, our method also applies to recursive functions provided that the recursion depth is independent of input data. Moreover, many infeasible paths, paths that cannot be executed regardless of the input data, will be excluded from the analysis and will therefore not affect the WCET estimation. Unlike existing compiler approaches, however, our symbolic execution does not terminate unless loop bounds are statically known.

2.3. Discussion

Ideally, for each combination of input data, the goal is to eliminate all infeasible program paths. The domain of values we use, however, makes it sometimes impossible to correctly analyze mutually exclusive paths. Consider for example the statements below where b is *unknown.*

```
if (b < 100)
  fun1()
if (b > 200)
  fun2()
```

Variable b will be *unknown* in both conditions, forcing the simulation of four paths even if `fun1()` and `fun2()` are mutually exclusive and the number of feasible paths are three. This can overestimate WCET because we only distinguish between known and unknown values. However, our approach does not hinder us from extending the domain of values to, e.g., intervals. This would handle the mutually excluding paths above. However, our implementation of the method, which we evaluate in this paper, uses the simpler domain. Nevertheless, we will discuss the implications of using other domains in Section 6.

As mentioned above, store instructions with an unknown reference address need special treatment. To efficiently handle these unpredictable accesses, our method identifies all data structures that are accessed with unpredictable memory instructions as *unpredictable data structures.* These are mapped by the linker into a memory area which can only return unknown values. This means that unpredictable accesses can be safely ignored. Predictable stores which access unpredictable data structures are not permitted to change the memory and predictable loads from unpredictable data structures will always return *unknown*, thus assuring a safe estimation of the WCET with no added cost in simulation time.

We identify the unpredictable data structures by letting the simulator output a list of all unknown stores it encounters. With the help of a source-code debugger, it is possible to manually connect each store instruction to a data structure which is marked as unpredictable. Eventually, the linking phase is redone to properly map the marked data structures to the 'unknown' area of memory. After this step, a correct estimation may be done. The approach mentioned above can be used for statically allocated data structures only. For unpredictable dynamically allocated data structures we do not yet have any solution. A further discussion about predictable and unpredictable data accesses can be found in (Lundqvist and Stenström, 1999).

A key problem with the simulation approach in this section is the explosion in number of paths to be simulated. If a loop iteration has n feasible paths and the number of loop iterations is k, the number of paths to simulate is n^k. Fortunately, good heuristics exist to drastically reduce the number of paths we need to simulate. We have used a path merging strategy, which forms the basis of our WCET method to be presented in the next section.

3. The WCET Method

To reduce the number of paths that have to be explored during path analysis, we apply a path-merge strategy. This reduces the number of paths to be explored from n^k down to n for a loop containing n feasible paths in each iteration and doing k iterations. In each loop iteration, all n paths are explored but in the beginning of the next iteration all these n paths are merged into one. Thus, the number of simulated paths is less than or equal to n. We first describe how the merging operation is performed in Section 3.1. This operation is used in the WCET algorithm which we present in Section 3.2. Finally, in Section 3.3 we discuss how we have implemented the method with a reasonable time complexity.

3.1. Merging algorithm

In order to understand how the merging of two paths is done, consider again the example program in Figure 1. In the second iteration of the outer loop, when the simulator encounters the unknown conditional branch on line 4, two paths can be merged. When merging, the long path, i.e., the one with the highest WCET, is kept and the short path is discarded. However, to make a valid estimation of the worst-case execution path throughout the program execution, the impact of the short path on the total WCET must be taken into account. For example, variable `sum` can be assigned different values in the two paths. Therefore, all variables whose values differ must be assigned the value *unknown* in the resulting path of the merge operation.

Formally, the algorithm views a path, p_A, as consisting of a WCET for the path, $p_A.wcet$, and the system state at the end of the path, $p_A.state$. The state of a path is the partial result of the execution including, e.g., the content of all memory locations, registers, and status bits that can affect the WCET in the future. In order to compute the state of the path resulting from the merging of several paths, the system states of the merged paths are compared and *unknown* values are assigned to locations whose values differ. We denote this operation as the *union* operation of path system-states.

The merging algorithm is described in Algorithm 1. It creates a new path p_C from two paths p_A and p_B. The program counter, which must not differ in the two paths merged, is copied to the new path. The new WCET is the maximum of the WCET of the two original paths. Finally, system states of the merged paths are unioned. The union operation between values is defined as:

$$a \cup b \Leftrightarrow \begin{cases} unknown & \text{if } a = unknown \text{ or } b = unknown \\ unknown & \text{if } a \neq b \\ a & \text{otherwise} \end{cases}$$

Algorithm 1 Merging two paths p_A and p_B creating p_C.

{PCT = program counter, CC = condition code}
Require: $p_A.state.PCT = p_B.state.PCT$
$p_C.state.PCT \leftarrow p_A.state.PCT$

$p_C.wcet \leftarrow \max(p_A.wcet, p_B.wcet)$

$p_C.state.CC \leftarrow p_A.state.CC \cup p_B.state.CC$
for all registers $R[i]$ **do**
  $p_C.state.R[i] \leftarrow p_A.state.R[i] \cup p_B.state.R[i]$
end for
for all memory positions a **do**
  $p_C.state.MEM[a] \leftarrow p_A.state.MEM[a] \cup p_B.state.MEM[a]$
end for

3.2. WCET algorithm

In order to implement the WCET simulation technique and the merging algorithm, one important issue is in which order all the paths should be simulated. Consider a loop with two feasible paths in each iteration. In order to merge these paths, they must have been simulated the same number of iterations. To accomplish this, the WCET algorithm needs loop information from the control flow graph of the program.

The algorithm (see Algorithm 2) starts the simulation from the beginning of the program. Whenever an unknown conditional branch is found the simulation is stopped and the algorithm selects as the next path to simulate the path that has made the least progress, *a minimum progress path.* If this path is not unique, all paths that have made the same progress and are at the same position in the program, *equal progress paths*, are merged into one before the simulation is continued. The progress of a path is a record of how many times the simulation of that path has passed each loop header and entered each function, as well as how far the simulation has proceeded in the current loop iteration, as dictated by the program counter.

By always selecting the path that has made least progress, the algorithm makes it certain that all paths in a loop iteration are simulated before a new iteration begins. In fact, merging will occur every time two paths that have made equal progress meet at the same position in the program. This makes an exponential growth of the number of paths impossible.

3.3. Time-complexity considerations

The time complexity of this method depends on how many paths need to be simulated, how often merging is done and how fast the actual merge operation is. How often the merging should be done involves an important tradeoff. If merging is done too often, it is likely that there will be fewer paths to simulate. However, when

Algorithm 2 WCET algorithm handling merge.

```
{A is set of active paths, C completed paths, ∅ = empty set, \ = set minus}
A ← ∅, C ← ∅

p ← starting null path
Simulate(p)
if p reached end of program then
  C ← C ∪ {p}
else {p reached an unknown conditional branch}
  A ← A ∪ {p}
end if

while A not empty do
  p ← minimal progress path in A
  A ← A \ {p}
  for all paths q with equal progress as p do
    A ← A \ {q}
    p ← merge(p, q)
  end for
  {Path p ends with a branch forcing a split}
  for each possible branch target i do
    p_i ← copy of p
    Simulate(p_i) along target i
    if p_i reached end of program then
      C ← C ∪ {p_i}
    else {p_i reached an unknown conditional branch}
      A ← A ∪ {p_i}
    end if
  end for
end while

wcet ← max_{p∈C} p.wcet
```

doing a merge, information can be lost because variables whose values differ are assigned *unknown*. This may result in infeasible program paths not being eliminated which in turn increases the number of simulated paths. Thus, merging less often can actually lead to a fewer number of simulated paths.

The algorithm we have implemented merges as often as possible. This makes an exponential growth of the number of paths impossible, e.g. in loops. Typically, each unknown branch found during the simulation would lead to the creation of yet another path, which later would result in an additional merge operation. The total number of paths simulated, as well as the number of merge operations, can in some cases grow in proportion to the number of loop iterations done in the program. One example is a loop with one unknown exit condition. The simulation of this loop would produce one new path (the exit path) to simulate each iteration. All these paths would, unless they reach the end of the program, be merged resulting in an equal number of merge operations.

For each merge operation, one must union the content of all registers and memory locations. This might be a quite slow process if the amount of memory is large. However, it is possible to speed up this operation considerably by utilizing the fact that paths that are to be merged, have often shared a long history of execution before they got split up. By only recording changes made to the system state since the time where the two paths were created, it is possible to quickly identify the parts of memory where the two system states differ. As an example, suppose that the system we model contains 1 Mbyte of main memory. Then, one can divide this memory into small fixed size chunks (say 512 bytes each) and each path only keeps the accessed chunks in its system state. In this way, only a few chunks of memory need to be compared during a merge operation.

4. Timing Analysis

The WCET algorithm in the previous section can estimate WCET for hardware platforms with fixed instruction execution times. Thus, an instruction-level simulation model extended to symbolically execute a program with unknown input data suffices. In this section, we extend this method to perform cycle-level symbolic execution in order to model the timing of high-performance processors employing multiple-issue instruction execution and instruction and data caching.

In order to update the WCET properly during simulation, the simulator must of course be extended to model the timing of caches and pipelines. In the context of caches, the simulator must model the impact of cache misses on the execution time. And in the context of pipelines, the simulator must account for the impact of structural, data, and control hazards (Hennessy and Patterson, 1996) on the execution time. With this capability, it is possible to make an arbitrarily accurate estimation of the WCET of a given path through the program.

A critical issue is how to carry out a merge operation. To do this, the method must estimate the impact of the system state on the future execution time. Such state information is exemplified by the identity of the blocks contained in the caches, which affects future misses, and the resources occupied by an instruction, such

as data-path components and registers, which affects future structural and data hazards. The merge operation introduced in Section 3 must be extended to handle such state information, which we will refer to as *timing state*.

To merge the timing state, we could use the same general principle as used when merging the content of memory locations; for all locations where the timing state differs we assert a pessimistic value, such as *unknown* in the merged timing state. For example in the case of caches, *unknown* means that a cache block is invalid, and the unioning of two cache timing states makes all cache blocks whose identities (i.e., memory tags) differ invalid. We call this method the *pessimistic merge* since it can incur a severe pessimism in the estimation. Fortunately, a method will be presented in Section 4.1 that avoids this in many cases. We will concretely apply this method to a high-performance processor whose timing model is introduced in Section 4.2. We then explain in detail how the method is applied to model caching and pipelining in Sections 4.3 and 4.4, respectively.

4.1. Optimistic merge approach

Our approach to reduce the pessimism caused by merging, is based on the idea that if it was known in advance which partial path belongs to the worst-case path through the program, one could update the worst-case execution time with the execution time of that partial path and also choose the timing state of that path when merging and then discard the timing state of the other path. Then, no pessimism would be incurred on a merge operation. While it is not possible to know in advance which of the two paths belongs to the worst-case path, a good guess would be that the longer of the two belongs to the worst-case path. If we also estimate how big effect the timing state of the shorter path has on the future execution time, we can make sure whether it is correct to use the WCET of the longer path along with its timing state when merging two paths. This approach is formulated in the following algorithm where we assume that the worst-case execution times of the long and the short paths are $WCET_L$ and $WCET_S$, respectively.

1. Estimate the *worst-case penalty* ($WCET_P$) that the short path would incur on the future execution time that will not be incurred by the long execution path.

2. If $WCET_L \geq WCET_S + WCET_P$ then use the timing state of the long path and $WCET_L$ in the merge operation and discard the timing state of the short path.

3. Otherwise, the pessimistic merge approach must be used.

In order to make this algorithm useful, we must clearly define what we mean with timing state, worst-case penalty, and pessimistic merging in the context of caches and pipelining. We will make these definitions in the context of a high-performance processor that uses many realistic features and which is presented in the next section.

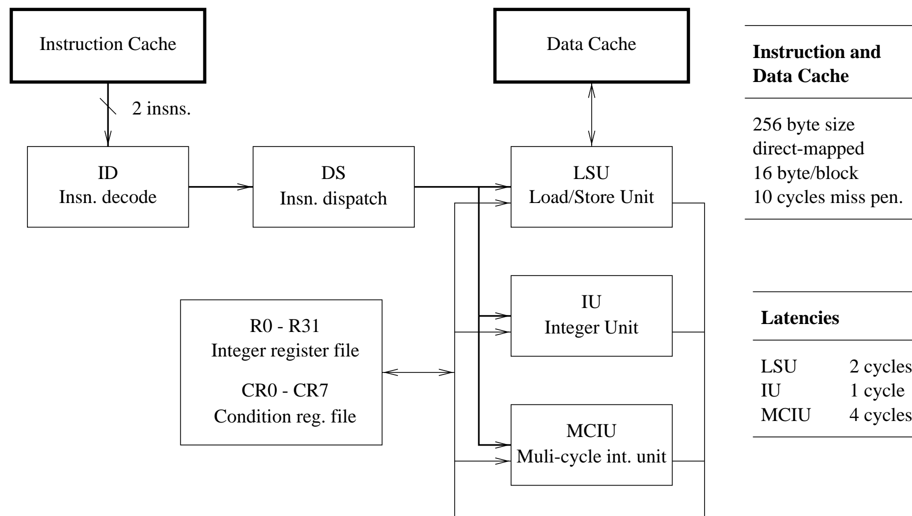


Figure 2. Example architecture.

4.2. Modeled architecture

The architecture used to demonstrate the timing analysis can be seen in Figure 2. It consists of a multiple-issue pipeline, capable of dispatching two instructions each clock cycle, and separate instruction and data caches. While the architecture is a subset of the PowerPC instruction set architecture — floating-point instructions are excluded — it nevertheless contains many of the key features critical for high-performance processors such as pipelining and caching.

Instructions are fetched from the instruction cache and put into the buffers in the *instruction decode* (ID) stage. From the decode stage, instructions are sent to the *dispatch stage* (DS) which in turn dispatches instructions to the three different functional units: the *load/store* unit (LSU), the *integer* unit (IU), and the *multiple-cycle integer* unit (MCIU).

At most two instructions in each cycle can be fetched from the instruction cache and put into the buffers of the ID-stage. A branch is handled ideally by not incurring any penalty, i.e., instructions will be fetched from the correct branch target in zero cycles. Instruction fetching will stall if the buffers of the ID-stage are full or if the fetch causes an instruction cache miss.

In the DS-stage, zero, one, or two instructions are dispatched each cycle in program order. An instruction can not be dispatched if it needs a resource which is currently busy or if an older instruction has not dispatched. Busy resources can be functional units (structural hazards) or registers (data hazards). An instruction reserves its destination registers and if later instructions use this register they are stalled until the first instruction completes and releases its destination register. Instructions in the pipeline are moved forward as soon as possible; if only one instruction is dispatched, instructions still advance so that at least one instruc-

tion is fetched from the instruction cache in order for the pipeline to dispatch two instructions during the next cycle.

All operands needed by an instruction are read from the register files or forwarded from another functional unit at dispatch time. There is one result-bus from each unit handling the write-back of data into the register file and the forwarding of data to another unit.

The load/store unit handles all loads and stores in an equivalent manner as far as timing is concerned. A load or store access that misses the cache causes a new entry (tag and block) to be allocated in the cache. Only one load or store is processed at a time. Normally, all LSU operations have a latency of 2 cycles, but if the access misses the data cache, the LSU unit will be busy during the fetching of the data including the data cache miss penalty.

The integer unit handles all single-cycle ALU operations in addition to the branch instructions. The multiple-cycle integer unit handles long-latency operations such as multiply and divide in addition to instructions involving any special purpose register in the POWERPC instruction set. All multiple-cycle instructions have a latency of 4 cycles. Each instruction thus has a fixed pipeline latency.

To keep the discussion simple, no contention exists when reading and writing in the register file or on the result buses. Also, the timing of all features external to the model (access to main memory) is assumed to be ideal, i.e., no contention exists between fetching data to the instruction cache and to or from the data cache. However, the model introduced accurately accounts for the overlap of simultaneous long-latency operations such as pipeline stalls and cache misses.

4.3. Instruction and data cache analysis

In the following, we first describe how to derive the worst-case penalty as introduced in Section 4.1 for caches. Also, in order to handle the case when it is not possible to discard the cache state of the short path, we need to define how to do a pessimistic merge between two cache states. Both instruction and data caches can be described and treated using the same principles. Therefore, we only treat how to handle instruction caching in this section. We will begin focusing only on direct-mapped caches before extending the method to handle set-associative caches.

Consider the timing state of two instruction caches, IC_L and IC_S, associated with the long and short path, respectively. A cache state is represented by an array of block identities, tags, showing how blocks are currently mapped in the cache (see the example in Figure 3a). To estimate the impact on the future execution time of discarding IC_S, we must consider all cases where IC_S may lead to a greater number of future cache misses than IC_L would. The worst case is found if we imagine that all cache blocks resident in IC_L but not found in IC_S will be needed in the future. We would then lose all these potential misses if we discard IC_S. To find the total possible loss of execution time when discarding IC_S, we go through all cache blocks and compare the tag in IC_S with the tag in IC_L and determine the number of entries where they differ. This number is then multiplied by the cache miss penalty to form the worst-case execution penalty.

Formally, the worst-case penalty for a direct-mapped instruction cache, $WCET_{IC}$, can be expressed as:

$$WCET_{IC} = P_{IC} \sum_{i=0}^{n-1} c(i)$$

where P_{IC} is the instruction cache miss penalty, n is the number of cache blocks, and $c(i)$ is defined by:

$$c(i) = \begin{cases} 1 & \text{if } IC_L(i) \neq IC_S(i) \text{ and } IC_L(i) \neq \text{invalid} \\ 0 & \text{otherwise} \end{cases}$$

where $IC_L(i)$ and $IC_S(i)$ is the cache tag for block i in the long and short path, respectively. We get a contribution to the worst case penalty if the tags differ between cache blocks in the long and short path, but not if the cache block in the long path is invalid. Then, it is impossible for the state in the short path to cause any additional misses in the future.

To derive the worst case penalty for set-associative caches, we need to first redefine the timing state of a cache. For an m-way set-associative cache with n sets, the timing state can be represented by a 2-dimensional array, IC, with size $n \times m$. The tag $IC(s,i)$ is then tag number i in set s. If we assume a LRU replacement policy, i.e., the least recently used block is replaced when a cache miss occurs, then the ordering of tags in a set reflects their relative LRU status, i.e., tag $i = 0$ belongs to the most recently used block and $i = m \Leftrightarrow 1$ to the least recently used block, which is the one to replace next.

Consider again the timing state of two instruction caches, IC_L and IC_S, associated with the long and short path, respectively. To estimate the impact on the future execution time of discarding IC_S, we must consider all cases where IC_S may lead to a greater number of future cache misses than IC_L would. This can happen if a cache block only is resident in IC_L and not in IC_S, or if it also is resident in IC_S but could get replaced earlier than in IC_L. A block, b, can be replaced earlier in IC_S than in IC_L if there exists another more recent block in IC_S that does not exist among the younger blocks in IC_L. In other words, all blocks younger than b in IC_S must also exist in IC_L and be younger than b in IC_L, in order for block b to survive in the worst case.

Formally, the worst-case penalty for a m-way set-associative instruction cache, $WCET_{IC}$, can be expressed as:

$$WCET_{IC} = P_{IC} \sum_{s=0}^{n-1} \sum_{i=0}^{m-1} c(s,i)$$

where P_{IC} is the instruction cache miss penalty, n is the number of sets, and $c(s,i)$ is defined by:

X = Invalid

	IC_L		IC_S
0	**0**	0	**0**
1	**1**	1	**1**
2	**2**	2	**2**
3	**19**	3	**3**
4	**20**	4	**4**
5	**21**	5	**5**
6	**X**	6	**6**
7	**X**	7	**7**
8	**X**	8	**8**
9	**X**	9	**9**
10	**10**	10	**10**
11	**11**	11	**11**
12	**12**	12	**12**
13	**X**	13	**X**
14	**30**	14	**X**
15	**31**	15	**X**

(a)

PL_L

Resource	Release Time	Stall Time
ID0	120	0
ID1	120	0
DS0	121	0
DS1	121	0
LSU	123	1
IU	122	0
MCIU	124	2
R0	124	2
R1	122	0
R2	123	1
R3	60	0
...		
R31	100	0
CR0	109	0
CR1	109	0
...		
CR7	109	0

PL_S

Resource	Release Time	Stall Time
ID0	100	0
ID1	100	0
DS0	101	0
DS1	101	0
LSU	113	11
IU	102	0
MCIU	94	0
R0	94	0
R1	102	0
R2	113	11
R3	60	0
...		
R31	100	0
CR0	89	0
CR1	89	0
...		
CR7	89	0

(b)

Figure 3. Example of (a) two instruction cache states and (b) two pipeline states.

$$c(s,i) = \begin{cases} 1 & \text{if there exists no block } k \text{ so that } IC_L(s,i) = IC_S(s,k) \\ & \quad \text{and } IC_L \neq \text{invalid} \\ 1 & \text{if there exists a block } k \text{ so that } IC_L(s,i) = IC_S(s,k) \\ & \quad \text{and } IC_L \neq \text{invalid} \\ & \quad \text{and if there also exists a block } g < k \text{ such that} \\ & \qquad IC_S(s,g) \neq IC_L(s,h) \text{ for all } h < i \\ 0 & \text{otherwise} \end{cases}$$

It should be noted that the $WCET_{IC}$ expression above is quite pessimistic. In reality, it may be impossible to replace combinations of blocks only in IC_S without replacing them in IC_L as well. The evaluation of the impact of this pessimism is outside the scope of this paper.

The pessimistic merging is done by invalidating cache blocks in the cache associated with the long path which are not already invalid and contribute to the worst case penalty. In this way, $WCET_{IC}$ can be reduced.

4.4. Pipeline analysis

Because it is well known how to take into account arbitrarily detailed timing models of pipelines in an architectural simulator (Pai et al., 1997), we will here focus on an approach to define the timing state needed to make an optimistic (or pessimistic) merge operation.

The pipeline will incur penalties caused by resource contention (structural hazards) and register dependencies (data hazards). For our pipeline model, resource contention and dependencies can be modeled using pipeline reservation tables, which record when each resource (pipeline stage or register) is released. An example is seen in Figure 3b. The ID- and DS-stages are divided into two sub-stages, ID0, ID1 and DS0, DS1, since each stage can hold two instructions at a time and each pipeline-stage buffer is treated as an individual resource. If two instructions are present in the pipeline stage, then both sub-stages are reserved. If only one instruction is present, then only ID1 (or DS1) is reserved.

During the simulation, the reservation table can be updated for each instruction at a time because all resources that the instruction requires are known. First, the instruction and data cache accesses are simulated. Then, the pipeline reservation table is updated to show when each resource is released. The current WCET of the path, i.e., the one used when merging, is updated to reflect the earliest point in time when it can be determined when each resource is released. Since this is the time when the last instruction is fetched we always keep WCET updated to this time. When a path through a program has been simulated the estimated WCET is the execution time of that path, defined as the time when the last resource is released in order to make a safe estimation.

Consider now the timing state of two pipelines, PL_L and PL_S, belonging to the long and short path, respectively (see the example in Figure 3b). We want to estimate the possible future effect on the execution time that PL_S may lead to compared to PL_L, and must in this case consider all future hazards that PL_S can lead to, which PL_L cannot lead to. For each pipeline stage and register we determine whether PL_S can make future instructions to stall for a longer time than PL_L would. The maximum difference in stall delay found is the worst-case penalty.

Formally, the worst case penalty for the pipeline, $WCET_{PL}$, is:

$$WCET_{PL} = \max_{r \in R}(s_S(r) \Leftrightarrow s_L(r))$$

where R is the set of all resources in the pipeline reservation table and $s_S(r)$ and $s_L(r)$ are the future stall times that resource r may cause based on the reservation tables of the short and long path, respectively.

To find the future stall time $s_S(r)$ (or $s_L(r)$) we simply imagine an instruction that uses only resource r, besides the ID and DS stages, and determine if it gets stalled if executed as the next instruction in the pipeline. However, since we model a dual-issue pipeline it is not enough using only one future instruction. To correctly handle the sub-stages of ID and DS, we must imagine two future instructions and choose the maximum stall time of the two instructions. An example of the stall times can be seen in Figure 3b. Using the stall times in the example we get: $WCET_{PL} = 10$.

If we are not allowed to discard PL_S, we must do the pessimistic merge instead. To reduce the worst case penalty for the pipeline we must identify the resources which contribute to $WCET_{PL}$. Then, by increasing the release-time entries in the pipeline reservation table for the long path, we can reduce $WCET_{PL}$.

4.5. Pessimistic merging

It is now possible to describe how the pessimistic merging approach can be done so that the instruction cache, data cache, and pipeline are taken into account altogether, and not as separate entities.

As described in Section 4.1 the merging is determined by the following optimistic merge condition, where $WCET_P$ is the worst case penalty and $WCET_L$ and $WCET_S$ are the WCET of the long and short path, respectively.

$$WCET_L \geq WCET_S + WCET_P$$

If this condition is true we can do an optimistic merge and discard the timing state of the short path and continue the simulation from the timing state of the long path. Otherwise, we must do a pessimistic merge. Before we describe how to do pessimistic merging for the complete system, we must define $WCET_P$, $WCET_L$, and $WCET_S$ for the entire high-performance system.

The worst case penalty, $WCET_P$, is the sum of the penalties for the instruction cache, data cache, and the pipeline:

$$WCET_P = WCET_{IC} + WCET_{DC} + WCET_{PL}$$

This is a safe, although pessimistic, estimate of the worst-case penalty that assumes that no overlap exists between future pipeline stalls, instruction, and data cache misses. Note, however, that this is only used as a merge condition; simulation of each path will accurately account for the overlap of long-latency events.

The pessimistic merge is done by modifying the timing state in order to reduce the worst-case penalty and make the optimistic merge condition true. We have chosen the following procedure:

1. Reduce the worst case penalty of the instruction cache ($WCET_{IC}$).

2. If the merge condition still is false reduce the worst case penalty of the data cache ($WCET_{DC}$).

3. If the merge condition still is false reduce the worst case penalty of the pipeline ($WCET_{PL}$).

When doing the pessimistic merge for the caches or the pipeline, it is not always necessary to reduce the worst-case penalty to zero. For example, when invalidating blocks in the instruction cache we can stop if the merge condition becomes true. In this way, as little pessimism as needed is introduced in the timing state. Fortunately, as we will see in the next section, optimistic merging is done in most cases.

5. Experimental Results

In order to evaluate the analysis accuracy of our cycle-level symbolic execution method, we have estimated the WCET of seven programs run on two different

Table 2. Characteristics of the programs used.

Name	Description
matmult	Multiplies 2 10x10 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
DES	Encrypts 64-bit data
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

processor architectures making it possible to evaluate the path and timing analysis capability in isolation. While both architectures support the same instruction set — a subset of POWERPC — the first one assumes that all instructions execute in a single cycle with a zero-cycle memory access time. In contrast, to focus on the accuracy of the timing analysis, the timing model of the second architecture corresponds to the system presented in Section 4. The timing parameters assumed for the caches and the latencies in the pipeline stages and functional units are depicted in Figure 2.

5.1. Methodology: Systems, test programs, and metrics

The WCET simulation method has been implemented by extending an existing instruction-level simulator, PSIM (Cagney, 1994-1996), with the capability of handling unknown values to allow for symbolic execution of all the programs. The WCET algorithm described in Section 3.2 has been added to control the path exploration and merging algorithms.

The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the programs. No optimization was enabled. The simulated run-time environment contains no operating system; consequently, we disabled all calls to system functions such as I/O in the programs.

An overview of the seven programs can be seen in Table 2. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *DES*, *jfdctint*, and *compress*.

In order to make a useful comparison how good estimates our method produces, we need to calculate the actual WCET of each program. The actual WCET has been determined by running the programs on the simulator with the worst case input data. This is straightforward to determine for all programs except *compress*, where the worst case input data is hard to find. Instead, a random sequence of 50 bytes has been used as input.

Another evaluation issue is that two of the programs, *fib* and *compress* actually have a termination condition that depends on input data. Therefore, we need to

Table 3. The estimated WCET using the ideal architecture.

Program	Actual WCET	Estimated WCET	Ratio	Structural WCET	Ratio
matmult	7063912	7063912	1	7063912	1
bsort	292026	292026	1	572920	1.96
isort	2594	2594	1	4430	1.71
fib	697	697	1	697	1
DES	118675	118675	1	119877	1.01
jfdctint	6010	6010	1	6010	1
compress	9380	49046	5.2	161161	17.2

bound the number of iterations to make WCET statically decidable. A common approach is to add manual annotations. This is not supported in our implementation. Instead, we have added an extra exit condition in the loops. In *fib* we have added the condition: $i \leq 30$ because we know that input data is always in this range. In *compress* we have bound an inner loop whose iteration variable is j, using the current iteration count, i, of the outer loop: $j \leq i$. This is a safe but pessimistic bound, but we have found it difficult to prove that a tighter bound could be used. The inner loop implements a secondary probe in a hash table and the number of iterations depends on unknown input data in a complex manner.

5.2. Path analysis results

In this section we evaluate to what extent our method manages to extract path information such as loop bounds and to what extent it manages to exclude infeasible paths from the analysis. We do this by using the idealized PowerPC architecture where each instruction executes in a single cycle.

In Table 3, we show three WCET numbers for each program: actual, estimated, and structural. The actual WCET is our measurement of the actual WCET as explained in Section 5.1. The estimated WCET is the WCET determined by the method and the structural WCET is the execution time of the longest structural path, including all infeasible paths, in the control flow graph of the program when using fixed bounds on the number of iterations of all loops. This number represents what a WCET method would estimate that does not eliminate any infeasible paths and uses fixed iteration bounds for all loops. The purpose of doing this is to analyze the capability of the method to eliminate infeasible paths. In the table, WCET is expressed in clock cycles and ratio is the estimated (or structural) WCET divided by the actual WCET.

For all benchmarks, except *compress*, we can see that the method succeeds in finding the actual WCET. In *compress*, the overestimation is caused by the inner loop. As mentioned previously, we bound this loop using the pessimistic condition $j \leq i$, but during a normal run, we have found that this inner loop is actually only

doing one single iteration. It should be mentioned that for *compress*, we do not know if the WCET we determined is the actual WCET. We suspect that the actual WCET we use is lower than the real one.

Two of the programs, *matmult* and *jfdctint*, have no infeasible paths at all, and only one path is simulated. In *DES*, however, there exist infeasible paths caused by data dependencies between different functions. These infeasible paths are eliminated by our method and only one path is simulated. In *bsort* and *isort*, all infeasible paths were not eliminated. Still, this does not lead to any overestimation, since all simulated infeasible paths are shorter than the worst-case path that the method finds.

If we take a look at the estimated structural WCET of the programs, we see that the WCET is grossly overestimated for *bsort*, *isort*, and *compress*. In *bsort* and *isort* it depends entirely on using a fixed iteration count for an inner loop which is normally bound by the current iteration count of the outer loop. This leads to an overestimation of a factor of two of the execution time for the loop and affects *bsort* more than *isort* because of the greater number of iterations done in *bsort*. In *compress*, there is a similar inner loop which is forced to have a fixed iteration bound again causing an overestimation of a factor of two. In addition, there exists a very long infeasible path that extends the structural estimate. As for *DES*, the tiny overestimation results from infeasible paths. As shown in Table 3, our method successfully manages to extract the loop bounds and eliminates the infeasible paths automatically.

A strength of doing the analysis on the instruction level has been revealed in *DES*. In the source code, one can find several conditional expressions which seem to indicate several possible feasible paths through the program. However, the compiler (gcc with no optimization enabled) automatically generates code without any branches for these conditional expressions and the resulting program has only a single feasible path. This is detected by our method.

5.3. Timing analysis results

In this section, we analyze how well our integrated path and timing analysis method manages to estimate the WCET of each program. We now assume the detailed timing model of the PowerPC architecture discussed in Section 4.2 with the timing parameters according to Figure 2. Table 4 shows three WCET numbers: actual, estimated, and conservative. Again, actual is the actual WCET we have determined whereas estimated WCET is the WCET determined by our method. Finally, conservative WCET corresponds to the estimated WCET when caches are turned off and each instruction proceeds through the pipeline one at a time. Finally, the two ratios shown correspond to estimated and conservative WCETs divided by the actual WCET, respectively.

Starting with all applications except *DES* and *compress* we note that our method manages to make an **exact** estimate of the actual WCET. This is somewhat unexpected even if these same programs were perfectly analyzed with respect to path properties. The expected added complication now stems from two sources: pes-

Table 4. The estimated WCET with caching and pipelining enabled. The ratios for the cache-all versions of *DES* and *compress* include the slowdown resulting from not caching accesses to unpredictable data structures.

Program	Actual WCET	D-cache Miss-rate	Estimated WCET	Ratio	Conservative WCET	Ratio
matmult	9715029	21.2 %	9715029	1	89741658	9.2
bsort	387331	3.0 %	387331	1	3474229	9.0
isort	3614	2.1 %	3614	1	38208	10.6
fib	1367	3.5 %	1367	1	12072	8.8
DES (cache pred)	323898	15.6 %	323898	1	1810204	5.6
DES (cache all)	323276	15.5 %		1.002		5.6
jfdctint	15276	8.0 %	15276	1	97213	6.4
compress (cache pred)	32235	53.4 %	103901	3.22	638265	19.8
compress (cache all)	27345	32.0 %		3.80		23.3

simistic merging of the timing state and the use of a data cache. We will now take a closer look on these sources in order to understand the reason for the good result.

The first reason for the good result when doing timing analysis is that no pessimistic timing merge was performed at all during the analysis. In *matmult*, *DES*, and *jfdctint*, no merge at all was needed since only one feasible path was simulated. In *fib*, all paths reached the end of the program before any merge was needed. Finally, in *bsort*, *isort*, and *compress*, only optimistic merges were needed, i.e., two paths that were to be merged always differed enough in length making it possible to discard the timing state of the short path and continue the simulation with the timing state of the long path.

The second issue which can cause overestimation is data caching. If the address of a data reference depends on unknown input data, this reference may in the worst case result in a miss which forces another block to be evicted from the cache. Thus, a safe estimate would be to charge two miss penalties to an unknown data reference. However, all applications, except *DES* and *compress*, contain only predictable data structures meaning that all data references are independent of input data. Thus, data caching is predictable and can be perfectly analyzed by our method.

DES and *compress* contain unpredictable data structures as defined in Section 2.3. In our method, we avoid the pessimism of charging two misses by using an alternative approach. Since unpredictable data structures are mapped into a special area of the memory, as discussed in Section 2.3, we also mark this area as non-cacheable. This is supported by most processors, since memory mapped I/O locations are in general not cached. During analysis we then know that an unknown reference will at most cause a single cache miss. Further discussion about unpredictable data accesses can be found in (Lundqvist and Stenström, 1999). For comparison purposes we have included two versions of *DES* and *compress*: one where only accesses going to predictable data structures are cached — called cache-predictable version — and one where all accesses are cached — called cache-all version. WCET estimation

has only been done for the cache-predictable versions. These numbers are shown in Table 4. The ratios shown for the cache-all versions of *DES* and *compress* are the estimated and conservative WCET for the cache-predictable versions divided by the WCET for the cache-all versions. It should also be mentioned that when data caching is enabled it is hard to find the worst case input data for the cache-all versions of *DES* and *compress*; the data addresses going to the data cache sometimes depend on unknown input data. We have not made any effort to address this problem.

As can be seen from Table 4, an almost perfect estimate of the WCET of *DES* is determined by our method in spite of unpredictable data structures. The numbers for the cache-all versions of *compress* and *DES* reveal to what extent the unpredictable data structures affect the estimation. In *DES*, only 0.6 % of all data accesses are directed to unpredictable data structures resulting in a slight overestimation. On the other hand, in *compress* 34 % of all accesses are directed to unpredictable data structures. By not caching these accesses we increase the execution time (and the overestimation) by 18 %.

The overestimation by a factor of 3.22 in *compress* is due to the inner loop as mentioned Section 5.2. This is lower than the factor of 5.2 previously found for the idealized architecture. The reason for this is that cache misses in the initialization part of the program makes the loop in the middle of the program a little less significant for the total program execution time.

To fully realize the importance of doing timing analysis, we can take a look at the conservative WCET in Table 4. We see that when treating all cache accesses as misses and permitting no pipelined execution we increase the overestimation of the WCET by a factor of between 5.6 and 10.6. In *compress*, for example, we find that the additional overestimation when doing no timing analysis is a factor of 6.1. Additionally, as we saw in the previous section, the overestimation when doing no path analysis (the structural WCET) is a factor of 3.3. Together, we get a factor of 20 in additional overestimation for *compress* when ignoring both path and timing analysis which clearly shows the importance of integrating accurate path and timing analysis.

6. Discussion

As the results indicate, to get a tight estimation of the WCET of a program, it is crucial to eliminate infeasible paths, especially in the presence of nested loops where a loop bound depends on the loop iteration variable of an outer loop. Of equal importance is an accurate timing analysis. The potential of detailed timing analysis is especially emphasized when optimistic merge is the common case. Then, as we saw in the previous section, it is possible to derive an exact estimate of the actual WCET. Also, as was illustrated by *DES*, *matmult*, and *jfdctint*, where there is only a single path through the program, this path can be simulated with an arbitrarily detailed timing model and will always give us an estimated WCET with no overestimation. Thus, by eliminating infeasible paths we can concentrate on the feasible ones, and make a more accurate timing analysis.

A big advantage of integrating the path and timing analysis can be seen when comparing with approaches where the path and timing analyses are kept separated. If an automatic path analysis is done first, we would need a way to represent the path information generated from the path analysis, and the timing analysis phase must be able to utilize this information. On the other hand, if the timing analysis is done first, we would be forced to work with fixed WCETs for blocks of statements when doing the automatic path analysis and WCET calculation. These problems are not present in our method, which does the path and timing analysis simultaneously.

Our cycle-level symbolic execution approach uses a domain where values can be either known or unknown. This simple domain performs remarkably well compared to more complex domains, e.g., based on intervals of values which is used in (Ermedahl and Gustafsson, 1997; Altenbernd, 1996). However, as we saw in Section 2.3, overestimations may sometimes arise for mutually exclusive paths. There is no inherent problem in extending our method to a more powerful domain. The result would be a slower simulation that needs more memory. Our choice of domain results in an additional 1 bit of memory for each 32-bit word of memory to hold the known/unknown status. An interval representation would need 2 extra words for each word of memory. Also, a more complex semantics would be needed, which would result in a slower execution of each instruction. On the other hand, the more complex domain might be preferable for some applications, if it manages to cut more infeasible paths and thereby gain speed and accuracy compared to our simple domain.

A more serious problem with our simple domain is that if all exit conditions of a loop is input data dependent, we get a completely unknown upper bound on the number of iterations in the loop, and our WCET algorithm will not terminate. This can be detected by using some heuristics or by user interaction. In these cases, we must add a manual annotation or add a known exit condition by modifying the loop condition in the program. For example, the loop in the program below, where b is unknown input data, will never do more than 100 iterations regardless of b. This fact cannot be represented with our simple domain, and during the simulation the loop will get a completely unknown exit condition, forcing us to add annotations or modify the program.

```
if (b < 100)
  for (i = 0 ; i < b ; i++)
    sum = sum + i;
```

The problem is caused by the simple domain of values. A similar problem can also arise when merging. The union operation used when merging may cause information needed to bound a loop to be lost. If this happens, we are also forced to annotate or change the program.

Our method uses the same basic idea regarding path analysis as the method used by Altenbernd in (Altenbernd, 1996), but he did not integrate any timing analysis. His method uses precalculated execution times on each basic code block and uses this to prune paths during the path exploration. However, in some cases, the

method suffers from complexity problems. We use a path merging strategy instead, which guarantees a manageable number of paths, and also makes it possible to integrate the timing analysis.

7. Related Work

We will finally put our method in context of recent work on path and timing analysis methods from the open literature.

Historically, one of the first approaches to provide path information was to require it from the programmer in terms of annotations (Puschner and Koza, 1989). This, of course, requires a considerable programming effort and is error-prone. In terms of automated approaches to path analysis two similar methods were recently proposed that rely on symbolic execution. Ermedahl and Gustafsson (Ermedahl and Gustafsson, 1997) use a path analysis approach that associates ranges with unknown input data and can use this domain to exclude infeasible paths automatically at the source-code level. Similar techniques that also use symbolic execution to perform timing analysis were proposed by (Chapman et al., 1994; Altenbernd, 1996). They estimate the WCET of programs with fixed execution times for basic code blocks. Like our method, they use symbolic execution but do not apply it on the cycle level. This prevents them from deriving useful estimates of WCET for high-performance processors where instruction execution-times depend on complex interactions between low-level long-latency events such as pipeline stalls and cache misses.

Timing analysis approaches for high-performance processors has gained a lot of attention recently (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Lim et al., 1995; Kim et al. 1996; Lim et al. 1998; Healy et al., 1995; White et al., 1997; Alt et al., 1996; Ferdinand and Wilhelm, 1998). Three quite different techniques are used in these approaches.

Integer linear programming and constraint solving is one approach (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Theiling and Ferdinand, 1998). It expresses WCET as a function of constraints on number of times specific paths are executed and the number of long-latency events that result from the execution of each path. Fixed-cycle penalties of cache misses and pipeline stalls are then associated with such events to derive an estimate of the WCET. Such methods provide accurate estimates only if exact path information is provided to express the constraints and if the execution time is a linear function in the number of long-latency operations. Unfortunately, accurate estimates have only been demonstrated for pipelined processors with caches where the overlap of long-latency events is not critical to the accuracy. Moreover, they have not yet demonstrated how to integrate it with an automated approach to provide path information.

The approach taken by (Lim et al., 1995; Kim et al. 1996; Lim et al. 1998) is to extend Shaw's timing schema (Shaw, 1989) to account for long-latency events in each basic code block. Unfortunately, unlike our method, their extension only approximates the overlap of long-latency events across basic blocks. In addition,

since they do not integrate the method with automated path analysis methods, it can result in considerable overestimations due to infeasible paths (Kim et al., 1996).

Abstract interpretation and dataflow analysis is another approach taken (Healy et al., 1995; White et al., 1997; Alt et al., 1996; Ferdinand and Wilhelm, 1998; Theiling and Ferdinand, 1998). Especially in the modeling of caches, dataflow analysis techniques have shown powerful to estimate a tight upper bound on the worst-case number of cache misses (Mueller and Whalley, 1995; Alt et al., 1996; Ferdinand and Wilhelm, 1998; Theiling and Ferdinand, 1998). Moreover, in combination with a reservation-table driven approach to model pipeline hazards, it is possible to model overlap between e.g. pipeline stalls and cache misses (Healy et al., 1995). Unfortunately, these methods also suffer from overestimations of WCET as they lack integration with automatic path analysis methods. In contrast, while we have shown that our cycle-level symbolic execution approach can make as accurate timing analysis as these methods, it also manages to extract important path information automatically. In this paper, we have demonstrated that it can model as aggressive multiple-issue processors as in (Lim et al. 1998) and as fancy cache organizations as in (White et al., 1997). In addition, we do not see any inherent limitation of the method to model multi-level caches as in (Mueller, 1997).

To the best of our knowledge, our method is the first that combines accurate path and timing analysis in one holistic approach.

8. Conclusions

In this paper we have presented a new method for estimating the WCET of a program. This method uses cycle-level symbolic execution to integrate path and timing analysis and thereby has the potential to do tight estimations by eliminating certain classes of infeasible paths and concentrating accurate timing analysis on the feasible ones.

Our results concerning the path analysis aspects show that many infeasible paths indeed are eliminated in the tested programs. In fact, for six out of the seven programs we managed to do a perfect estimation. We also studied how the impact of performance enhancing features in high-performance processors, such as pipelined multiple-issue execution, instruction and data caching, could be incorporated in the symbolic execution method. The critical issue to address is to formulate conditions for the merge operation. We demonstrated how this is done for pipelining and caching. In evaluating the accuracy of the timing analysis we found that we could make an exact estimate of the WCET, again for six out of the seven programs we tested. The overall effect of integrating path and timing analysis was shown to improve the WCET estimation a factor of twenty in some cases over a conservative method.

Acknowledgments

We are deeply indebted to Dr. Jan Jonsson of Chalmers for his constructive comments on previous versions of this manuscript. We would also like to acknowledge

the helpful comments given by people involved in the Swedish WCET Network (see `http://www.docs.uu.se/artes/wcet/`). This research is supported by a grant from Swedish Research Council on Engineering Science under contract number 221-96-214.

References

Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. 1996. Cache behavior prediction by abstract interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, pp. 52–66.

Altenbernd, P. 1996. On the false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pp. 102–107.

Cagney, A. 1994-1996. PSIM, a PowerPC simulator. http://sourceware.cygnus.com/psim/.

Chapman, R., Burns, A., and Wellings, A. 1994 Integrated program proof and worst-case timing analysis of SPARK Ada. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. K1–K11.

Ermedahl, A., and Gustafsson, J. 1997. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, pp. 1298–1307.

Ferdinand, C., and Wilhelm, R. 1998. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, pp. 16–30.

Healy, C., Whalley, D., and Harmon, M. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 288–297.

Healy, C., Sjödin, M., Rustagi, V., and Whalley, D. 1998. Bounding loop iterations for timing analysis. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pp. 12–21.

Hennessy, J., and Patterson, D. 1996. *Computer Architecture: A Quantitative Approach, 2ed.* Morgan Kaufmann.

Kim, S.-K., Min, S., and Ha, R. 1996. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 230–240.

Li, Y.-T., Malik, S., and Wolfe, A. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 298–307.

Li, Y.-T., Malik, S., and Wolfe, A. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 254–263.

Lim, S.-S., Bae, Y., Jang, G., Rhee, B.-D., Min, S., Park, C., Shin, H., Park, K., and Kim, C. 1995. An accurate worst case timing analysis for RISC processors. In *IEEE Transactions on Software Engineering*, 21(7), pp. 593–604.

Lim, S.-S., Han, J., Kim, J., and Min, S. 1998. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 334–345.

Lundqvist, T., and Stenström, P. 1999. Empirical bounds on data caching in high-performance real-time systems. Technical Report 99–4, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden.

Magnusson, P., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Moestedt, A., Nilsson, J., Stenström, P., and Werner, B. 1998. Simics/sun4m: A virtual workstation. In *Proceedings of USENIX98*, pp. 119–130.

Mueller, F. 1997. Timing predictions for multi-level caches. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 29–36.

Mueller, F., and Whalley, D. 1995. Fast instruction cache analysis via static cache simulation. In *Proceedings of 28th Annual Simulation Symposium*, pp. 105–114.

Ottosson, G., and Sjödin, M. 1997. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 47–55.

Pai, V., Ranganathan, P., and Adve, S. 1997. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *IEEE Technical Committee on Computer Architecture newsletter*, pp. 32–38.

Puschner, P., and Koza, C. 1989. Calculating the maximum execution time of real-time programs. In *Journal of Real-Time Systems*, 1(2), pp. 159–176.

Shaw, A. 1989. Reasoning about time in higher-level language software. In *IEEE Transactions on Software Engineering*, 15(7), pp. 875–889.

Theiling, H., and Ferdinand, C. 1998. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 144–153.

White, R., Mueller, F., Healy, C., Whalley, D., and Harmon, M. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 192–202.

Paper II

Timing Anomalies in Dynamically Scheduled Microprocessors

Reprinted from

Technical Report 99–5
Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden, April 1999
Submitted for publication.

Timing Anomalies in Dynamically Scheduled Microprocessors

Thomas Lundqvist and Per Stenström

May 11, 1999

Abstract

Safe and tight estimations of the worst-case execution time (WCET) of programs run on processors employing pipelining and caching is important when constructing high-performance real-time systems. Previous timing analysis methods have assumed that the worst-case instruction execution time necessarily corresponds to the worst-case behavior. We show that this assumption is wrong in dynamically scheduled processors. A cache miss, for example, can in some cases result in a shorter execution time than a cache hit. Many examples of such timing anomalies are provided.

A first contribution of this paper is to provide necessary conditions for when timing anomalies may show up and identification of what architectural features that may cause such anomalies. We also show that analyzing the effect of these anomalies with known techniques would result in prohibitive computational complexities. Instead we propose some simple code modification techniques to make it impossible for any anomalies to occur. These modifications make it possible to estimate WCET by known techniques. We use an existing WCET analyzer to evaluate how much pessimism the code transformations impose on some benchmarks. Our evaluation shows that the pessimism imposed by these techniques is fairly limited; it is less than 27 % for the programs in our benchmark suite.

1 Introduction

Estimation of an upper-bound on the execution time, called worst-case execution time (WCET), is important for highly dependable real-time systems. Because of pessimistic timing assumptions, WCET is often grossly overestimated which results in poor resource utilization, especially in real-time systems using high-performance processors with advanced pipelining and caching techniques.

WCET is typically estimated as tight as possible by analyzing the WCET of each path in the program—often in combination with heuristics to prune the number of paths

to analyze. Moreover, this analysis often proceeds from the first to the last instruction in each path. In doing this, one must take into account that the execution time (latency) of each instruction is not fixed; it can take one of many discrete values depending on input data. The way known methods deal with this problem [2–8] is to assume the longest execution time because the intuition is that this will always cause a conservative estimate of the WCET. For example, if the outcome of a cache access is unknown, a cache miss is assumed.

We show in this paper that this intuition is simply wrong for many high-performance processors using dynamic instruction scheduling. Because the instruction schedule depends on the execution time of each individual instruction, the scheduling of future instructions can actually cause a counter-intuitive increase or decrease in the execution time of the rest of the execution path. We will show many examples of such timing anomalies in the paper.

To find a safe estimate of the WCET in the presence of such anomalies, one would have to analyze the effect of all possible schedules resulting from a variable-latency instruction to find the instruction execution time that leads to the longest overall execution time. In general, if we have n variable-latency instructions along a path in the program, where each instruction may lead to k different future schedules, then, in the worst case, one must analyze k^n different schedules. We show that previously published analysis methods for cache and pipeline analysis [2–4, 6, 8] would result in prohibitive computational complexity for analyzing these anomalies.

This paper first identifies necessary conditions for when timing anomalies can show up in dynamically scheduled processors and what architectural features that may cause them. We then propose some simple code modification techniques that eliminate the existence of timing anomalies, thus enabling known WCET analysis methods to estimate WCET. The main idea exploited is to make program modifications that will guarantee that a future instruction schedule is not affected by a variable-latency instruction. We evaluate the amount of pessimism introduced on a number of benchmark programs by instruction-level simulation and a model of a dynamically scheduled processor. Our main conclusion is that the pessimism introduced by the modifications is fairly limited; it is less than 27 % for the programs in our benchmarks suite.

The rest of the paper is organized as follows. In Section 2, we first consider when and how timing anomalies show up in dynamically scheduled processors. In Section 3, we show why previous methods fail to handle these anomalies. The rest of the paper is devoted to our approach to handle the anomalies. We introduce the idea of program modifications in Section 4 which we evaluate experimentally in Section 5. Finally, we discuss our approach and also point out future directions of research in this area in Section 6, before we conclude in Section 7.

2 Timing Anomalies in High-Performance Processors

In this section, we will give examples of the timing anomalies present in dynamically scheduled processors. But first, we define necessary conditions that can lead to such anomalies. The term dynamically scheduled processors is often used to describe a processor for which instructions execute out-of-program-order. In the first section, a first contribution is that we show that it is not the out-of-order execution that is the central issue here. Rather, it is the order in which resources are allocated in the processor.

2.1 Timing anomalies: definitions and conditions

The execution time of an instruction can take one of many discrete values depending on input data. One example is a load instruction whose execution time depends on whether the address hits or misses in the cache. Another example is an arithmetic instruction whose execution time may depend on the operands. A common assumption is that if the worst-case instruction execution time is assumed, the WCET estimation will be safe. Throughout this paper, we define a timing anomaly as a situation when such assumptions do not hold. For clarity reasons, we will use the term *latency* meaning the instruction execution time. When we use the term execution time it will often mean the overall execution time of the program.

Consider the execution of a sequence of instructions. Let us study two different cases where we modify the latency of the first instruction. In the first case, we increase the latency by i cycles. In the second case, we decrease the latency by d cycles. Let C be the future change in execution time resulting from the increase or decrease of the latency. Then:

Definition 1 *A timing anomaly is a situation where, in the first case, $C > i$ or $C < 0$, or in the second case, $C < -d$ or $C > 0$.*

That is, if C is guaranteed to be in the interval: $0 \leq C \leq i$ in the first case or $-d \leq C \leq 0$ in the second case, we have no timing anomalies.

To model the instruction execution in a pipelined processor, one often uses a resource model. In this model, whenever an instruction that proceeds through a pipeline gets stalled, it is due to resource contention with another instruction that accesses a common resource or operand. Typical examples of resources are functional units and registers, but also buses, read and write ports, and buffers should be treated as resources if they can cause instructions to stall.

The resources that an instruction can use can be divided into *in-order* and *out-of-order resources*. In-order resources can only be allocated in program order to instructions. Out-of-order resources can be allocated to instructions dynamically, i.e., a new instruction can use a resource before an older instruction uses it according to some dynamic scheduling decision. Typical out-of-order resources are functional units that

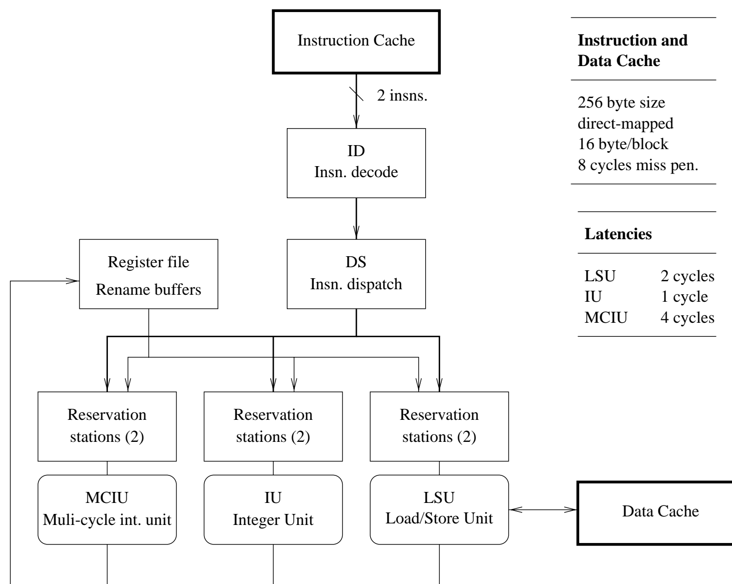


Figure 1: A simplified, yet timing-anomalous, PowerPC architecture.

service instructions dynamically (out-of-order initiation). An example of in-order resources is such registers which must be reserved in-order to guarantee that data dependencies in the program are not violated. It is now possible to state exactly a sufficient condition when a processor is free from anomalies:

Condition 1 *If a processor only contains in-order resources no timing anomalies can occur.*

To see why this condition is sufficient, consider a processor that only contains in-order resources. This means that two instructions can only use a resource in program order. If the completion of an instruction is postponed by i cycles, later instructions will also be postponed since they cannot allocate the resource before the first instruction. However, it is possible that future instructions will be postponed by less than i cycles if the new schedule becomes more compact, i.e., containing less idle time. The amount postponed cannot be less than 0 cycles however. Thus, C will be less than or equal to i and greater than 0. The same principle apply if an instruction is completed d cycles earlier. To conclude, if all resources are in-order no timing anomalies may occur.

2.2 Timing anomaly examples

If out-of-order resources are present, timing anomalies may occur. To see how, we will now study an architecture containing out-of-order resources and give examples of how

timing anomalies may occur.

The focus of our study will be the model of an architecture seen in Figure 1 based on a *simplified* PowerPC architecture containing no floating point units. A more realistic model is expected to contain more features that would result in out-of-order resource allocation. Our point is then that even for this simplified architecture, timing anomalies show up.

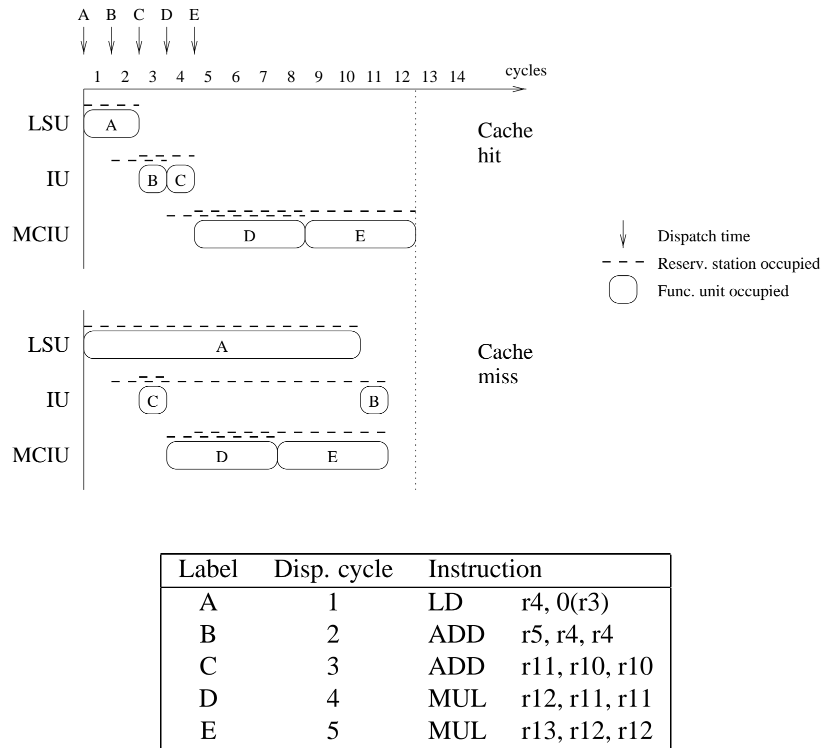
The architecture consists of a multiple-issue pipeline, capable of dispatching two instructions each clock cycle, and separate instruction and data caches. To implement out-of-order execution of instructions, each functional unit has two reservation stations. These can hold dispatched instructions before their operands are available. Register renaming is used to avoid unnecessary data hazards. Also needed, but not shown, is a completion unit with a reorder buffer, which completes instructions in-order by updating the register file from the renaming buffers.

All resources in the modeled processor are considered to be in-order resources except the integer unit (IU) and the multiple-cycle integer unit (MCIU) which are out-of-order resources. The load/store unit (LSU) often initiate execution in-order to preserve ordering of memory accesses so we also treat it as in-order here. The out-of-order resources, IU and MCIU, make timing anomalies possible as we will demonstrate in three examples: one showing that a cache hit may be worse than a cache miss, another showing that the miss penalty can be greater than expected, and a third showing a possible domino effect when executing loops.

2.2.1 Anomaly 1: A cache hit can result in worst-case timing

The first example presents a case where a data cache hit causes an overall longer execution time than a data cache miss. Consider the table in Figure 2, which shows a sequence of instructions (A-E) and in which cycle they are dispatched. The instructions represent the use of different functional units: the `LD rd,0(ra)` instruction uses the LSU, the `ADD rd,ra,rb` uses the IU, and the `MUL rd,ra,rb` uses the MCIU. Register `rd` is the destination register and `ra` and `rb` are the source registers. The registers create data dependencies and thereby an ordering between instructions. To simplify the discussion of the examples we focus only on the functional units and their reservation stations. We assume that the instructions are dispatched according to the relative times seen in the instruction table in Figure 2 although in reality, on a dual-issue pipeline, additional instructions would be needed to make the instructions dispatch according to the example.

The diagram in Figure 2 shows when each functional unit is busy executing an instruction. Also shown as horizontal dashed lines is when the reservation stations are occupied. At the top, arrows indicate when each instruction is dispatched to the reservation stations. Two cases can be seen, one when the load address hits in the data cache and one when it misses the cache.



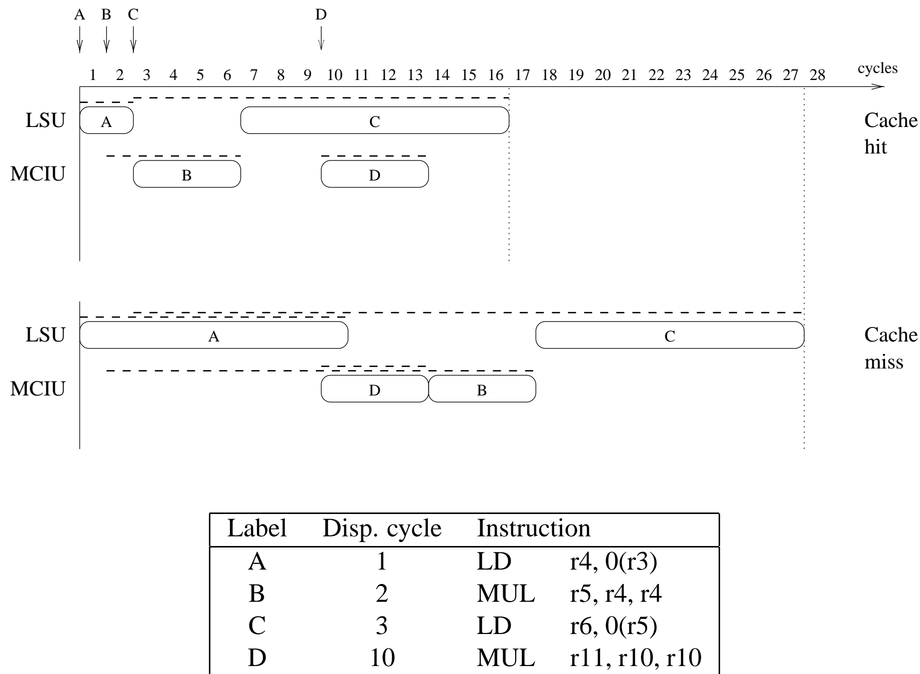
Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	ADD	r5, r4, r4
C	3	ADD	r11, r10, r10
D	4	MUL	r12, r11, r11
E	5	MUL	r13, r12, r12

Figure 2: An example when a cache hit causes a longer execution time than a cache miss.

If the load address hits in the cache then the `LD` instruction executes for 2 cycles and can forward its result to instruction B which can start executing in cycle 3. Here, we assume that B gets priority over C since B is older. Thus, C must wait for B. On the other hand, if the load address misses in the cache then the `LD` instruction executes for 10 cycles and the execution of B will be postponed. This means that C can start executing in cycle 3, one cycle earlier than in the cache hit case. This will make D and E execute one cycle earlier as well, leading to an overall reduction of the execution time by 1 cycle in the cache miss case. In this case, the anomaly is made possible due to the IU being an out-of-order resource permitting B and C to execute out-of-order.

2.2.2 Anomaly 2: The miss penalty can be higher than expected

The second example shows that the overall penalty in execution time due to a cache miss can be higher than the normal cache miss penalty. Consider the instruction sequence in Figure 3. The first instruction is a load instruction which can either hit or miss in the



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	MUL	r5, r4, r4
C	3	LD	r6, 0(r5)
D	10	MUL	r11, r10, r10

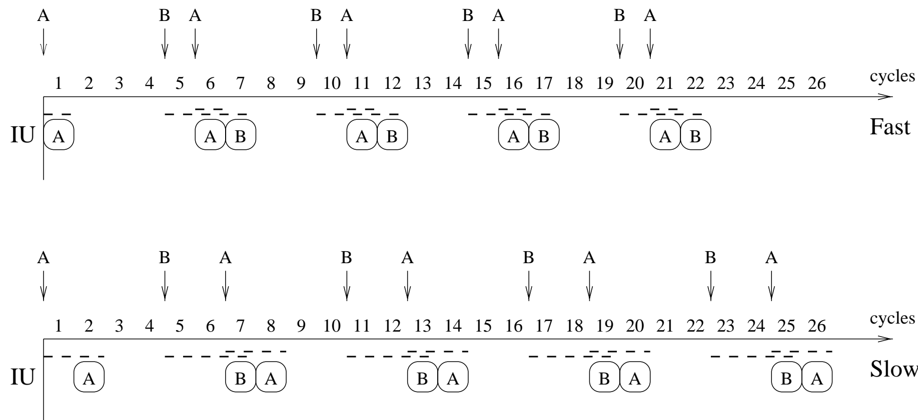
Figure 3: An example when the cache miss penalty is higher than expected.

cache. We assume that the second load instruction (C) always misses. The first three instructions: A, B, and C, depend on each other and must execute one at a time. In the cache hit case all instructions will execute as soon as possible. The last instruction, D, will not interfere with the execution of the other instructions.

If the first load experiences a cache miss, the execution of B will be postponed. In this unfortunate case, instruction D has already started when B becomes eligible for execution and B will be further postponed. The result of this is that instruction C will finish executing 11 cycles later in the cache miss case compared to the cache hit case. This is greater than the normal cache miss penalty of 8 cycles. In this case, the anomaly is due to the MCIU being an out-of-order resource, which allows instruction B and D to execute in arbitrary order.

2.2.3 Anomaly 3: The impact on WCET may not be a fixed constant

We saw in the previous example how the total penalty of a cache miss can be increased due to changes in the instruction schedule. However, it is bounded by a constant value. We will now show an example when the increase is not necessarily limited by a constant value, but can be proportional to the length of the program. This means that a small



Label	Disp. cycle	Execute cycle	Instruction	
A	$E_A + 5$	Immediate	ADD	r4, r3, r3
B	$D_A + 4$	$D_A + 6$	ADD	r11, r10, r10

Figure 4: Example of domino effect.

interference in the beginning of the execution may contribute with an arbitrarily high penalty to the overall execution time.

Consider the instruction sequence in Figure 4. The two instructions A and B constitute the body of a loop doing a number of iterations. The delicate execution scenario shown here demands special requirements on the dispatch and execute cycles. Therefore, the table entry for the dispatch cycle and the additional table entry for the execute cycle show the dispatch and execute cycle relative to a previous instruction. By E_A we mean the cycle when A executed in the previous iteration of the loop. By D_A we mean the cycle when A was dispatched in the current loop iteration.

The two different scenarios shown in Figure 4 are the result of dispatching and executing the two instructions A and B repeatedly according to the dispatch and execute cycle rules starting from two different executions of the first A instruction. In the fast case, instruction A in the first iteration executes immediately when it is dispatched. In the slow case, we imagine that it gets delayed one cycle because of a dependency with an earlier instruction. This delay in the beginning is enough to cause a domino effect that will delay the execution of A by one cycle in each iteration. The total penalty on the execution time, caused by the small delay of A in the beginning, will be k cycles if the loop does k iterations. In the slow case, we assume that the old B instruction gets priority over the new A instruction in each iteration.

In summary, we have shown three examples when timing anomalies may show up

in dynamically scheduled processors. These anomalies were possible due to the presence of out-of-order resources. The first two examples show that worst-case instruction execution assumptions may result in optimistic estimates of the WCET if the future scheduling is not taken into account. It is not difficult to construct other instruction sequences where similar anomalies appear. While the last example shows a presumably rare event, it emphasizes that it may not be safe to make assumptions regarding timing on the instruction level.

3 Impact of Timing Anomalies on WCET Methods

In the previous section we have seen that timing anomalies may occur in dynamically scheduled processors. To correctly estimate the WCET, one would have to consider the effect all variations in instruction execution times have on the possible instruction schedules. We will now consider the problems that arise if we want to perform accurate pipeline analysis for dynamically scheduled processors and how previous methods fail to handle these problems. To simplify the discussion, we will use the following definitions:

Definition 2 *The current pipeline state is the current state of the pipeline timing model. It describes which instructions are currently executing in the pipeline and the current resource allocations.*

Definition 3 *The current cache state is the current content of the cache timing model. It consists of the cache tag memory, i.e., the identification tags of the current blocks in the cache.*

Consider first a program containing only a single feasible path. The WCET is then the longest execution time of the instruction sequence along this path. Assume that the sequence contains n variable-latency instructions with unknown latencies, but we know that each instruction can have k different latencies. Then, we must for each variable-latency instruction find the latency that causes the longest overall execution time. To be safe, we must examine k^n instruction schedules because the execution of each variable-latency instruction may cause k schedules of all succeeding instructions.

In general, analyzing all k^n combinations is not feasible and another approach is needed. Normally, timing analysis methods rely on the possibility of making safe decisions locally at the instruction or basic block level. That is, a pessimistic choice is always made at this level. Unfortunately, due to the anomalies, we cannot make a local safe decision. Consider a partial sequence of the instructions, e.g., a basic block, containing a variable-latency instruction. When simulating the execution of this partial sequence in the pipeline we may end up with k different pipeline states. To be safe, we must then choose the pipeline state that will give us the longest overall execution time. But this is impossible without knowledge of the whole instruction sequence.

All previously presented methods for doing cache and pipeline analysis [2–8] perform the pipeline analysis by first looking at each instruction or basic block and then combining the WCET of all these entities into a total WCET for the whole program. While none of these methods are designed to handle dynamically scheduled processors, they nevertheless rely on being able to make local safe decisions when regarding variable-latency instructions. For example, in [2, 8] the cache analysis is done first and then later used in a pipeline analysis step. Whenever it is not possible to classify a cache access as a hit or a miss, it is conservatively assumed to be a miss. This may lead to an optimistic estimation as we have seen in the first anomaly example according to Figure 2.

Consider next a program containing several feasible paths. The WCET is then the maximum WCET found among all paths and to find the WCET we would have to examine all paths in the program. This is in general not feasible and timing analysis methods again rely on the possibility of making local safe decisions to reduce the complexity. When analyzing a small section of the program, e.g., a loop, the longest path in this section is chosen before doing the analysis of the rest of the program. Unfortunately, due to the anomalies, it is not possible to make local safe decisions. Assume that the small section contains l different paths. When simulating the execution of the different paths in the pipeline we may end up with l different pipeline states, leading to the same problem as for the variable-latency instructions. It is not possible to know which pipeline state (path) that gives us the longest overall execution time.

An example of when local decisions are used to reduce the path complexity is the prune operation used in [4, 5]. It is used to discard some combinations of basic blocks that will execute in shorter time than another combination of blocks found. To make this pruning decision, one must know how the execution of some basic blocks will influence the execution of other parts in the program. Due to, e.g., the domino anomaly (Figure 4), this can be very hard or impossible. The same problem exists in [2] where the longest path is chosen in each iteration of a loop.

To conclude, when doing timing analysis in the presence of timing anomalies, it is not possible to make local safe decisions, i.e., safe choices between different pipeline states that an unknown event may give rise to. There are two typical cases. The first case is how to know which latency to choose for a variable-latency instruction and the second case is how to select the longest path in a small section of the program in order to reduce the path complexity.

In spite of these problems, we will in the next section show two new approaches that can make it possible for previously published timing analysis methods to handle dynamically scheduled processors.

4 Timing Analysis Methods for Dynamically Scheduled Processors

In this section, we will present two new approaches to estimating the WCET of a program running on a dynamically scheduled processor where we might experience timing anomalies. Both approaches can be used together with previously published timing analysis methods. We first present the serial-execution method, a pessimistic but safe method to handle architectures with timing anomalies. After this, we present our new method based on program modifications—by modifying the program we make it possible for timing analysis methods to rely on safe local decisions. At the end of this section, we present a case study of how the program modification method can be used together with our previously published method based on cycle-level symbolic execution [6]. We only focus on the basic pipeline and instruction and data cache analysis.

4.1 Pessimistic serial-execution method

A straight-forward way to make safe estimations for architectures containing anomalies is to use the pessimistic serial-execution estimate. This means that we model all instructions as being executed non-pipelined in the functional units. That is, we sum all instruction latencies. In addition to this, we add the miss penalties for all instruction and data cache misses. We now formulate a claim that needs to be proven although intuitive in nature.

Claim: The WCET corresponding to a serial execution of the instructions, assuming their worst-case latencies, is always higher than the WCET corresponding to any pipelined execution of the same instruction sequence.

Proof: Instructions can not execute slower than non-pipelined since this would mean that some functional units are idle sometime. This can not be true since instructions are always available for execution. The only possibility for an instruction to stall is cache misses which we add separately.

The serial-execution estimate will be safe but not very tight. A big advantage, however, is that unknown events in the system are handled in a safe way. They can not lead to a greater execution time than the one estimated for serial execution.

4.2 Program modification method

The serial-execution method is very pessimistic. If we want a tighter estimated WCET we must model the pipelined execution accurately and deal with the problem of timing anomalies. One way of accomplishing this is to modify the program so that we can rely on safe local decisions. In short, we want to make sure that the following conditions are true:

1. All variable-latency instructions that have an unknown latency must when simulated still result in a predictable pipeline state. Also, we must make sure that the worst-case latency is used for the instruction. In addition, other unknown events such as unknown instruction cache accesses must also result in a predictable pipeline state.

2. If the number of paths in a small section of the program is being reduced by selecting the longest one or discarding the shortest ones, then the state of the pipeline and the caches at the beginning and end of the paths must not differ when comparing them.

To fulfill the first condition one must force an in-order resource use when executing the variable-latency instruction. Also, the pipeline state must be predictable before allowing out-of-order resource use again. The way to accomplish this is highly architecture dependent. Unfortunately, no support for in-order resource scheduling is present in processors today, but other instructions may be used for this purpose. For example, in the POWERPC architecture, there is a memory synchronization instruction called `sync`, which inhibits further dispatching until the `sync` instruction completes. This instruction can be used as a way to force serialization together with a variable-latency instruction.

If one `sync` is placed after the variable-latency instruction then the pipeline state will be known afterwards. If one `sync` is placed before the variable-latency instruction we will know for sure that the instruction will execute in-order and the maximum latency will be the worst-case latency. Also, for other unknown events, like an unknown instruction cache access, we can also use the same method to make the pipeline state predictable. In the rest of this paper we will assume that an instruction such as `sync` exists.

To fulfill the second condition above we can again use the `sync` instruction to handle the pipeline state. For example, by placing such an instruction at the end of two paths, the pipeline states in the two paths are made equal to each other. The state of caches is more tricky to handle. One need to set the state of the caches corresponding to the two paths being compared equal to each other. How this can be done is also highly architecture dependent. There are several options available:

1. One can invalidate all blocks in the caches. This should be possible in almost all processors.

2. One can invalidate only the blocks that differ in the two caches. This requires support for invalidation on the block level.

3. One can replace the blocks that differ with blocks that will be needed in the future by preloading blocks into the caches. This requires support for explicitly loading blocks into a cache.

The first option of invalidating the complete contents of the caches is obviously not an attractive solution since the performance will most probably become poor. This is true also for the second option since each invalidate operation will in many cases cause an additional cache miss later on. The third option is the most promising one but requires special instructions to preload the cache. An example of such instructions is the instruction and data cache block touch instructions (`icbt` and `dcbt`) found in the POWERPC architecture (but not all processors implement them).

When preloading blocks, it is best to preload a block that will be needed somewhere along the worst case path. Then, no unnecessary pessimism is added due to additional cache misses. To automatically calculate the best blocks to preload is a complex issue, which we do not investigate further in this paper. In the experimental evaluation, we derived this information manually (see Section 5).

When safe local decisions can be made, one can use previously published timing analysis methods when estimating the WCET for programs running on a dynamically scheduled processor. However, to really use one of these methods one must also specify at which points in the program a particular method relies on safe local decisions. Furthermore, the timing model used by the method must be extended to model the dynamically scheduled pipeline. If this is possible and how it is done for each individual method is beyond the scope of this paper. Yet, in the next section, we will describe how it is done for our previously published method based on symbolic execution.

4.3 Case study: Cycle-level symbolic execution method

We will now take a closer look at how the program modification method can be used together with our previously published WCET estimation method [6], based on cycle-level symbolic execution. We start with a brief description of our timing analysis method.

Our WCET estimation method is based on a cycle-level architectural simulator, which can be seen as an instruction-level simulator together with a detailed timing model of the architecture. By using such a simulator, it is possible to get tight estimations of the WCET for single paths through the program. However, in order to estimate the WCET for the whole program, the simulator has been extended to handle unknown data values and to enable symbolic execution of programs. In addition to exploring all feasible paths in the program, many infeasible (non executable) paths are also eliminated. The number of paths to explore can easily become prohibitive. Therefore, a path merge strategy is used to reduce the number of simulated paths. Typically, if a loop contains two feasible paths, these will be merged into one path before starting a new iteration, thereby reducing the number of paths to simulate to at most two in this case.

In order to estimate the WCET for a dynamically scheduled processor we must first attach the simulator to a timing model which accurately model the execution of instruc-

tions in the pipeline including the instruction and data caches. Then, we must modify the program to be able to make safe local decisions. This is done by first estimating the WCET of the unmodified program. In this process, we identify all places in the program where the analysis needs to make local decisions. In our case, this is when variable-latency instructions with unknown latency is found and whenever a merge operation is done during the analysis. At all identified places in the program, modifications are applied in order to make all the local decisions safe, i.e., `sync` instructions are inserted to handle pipeline states that differ, and all blocks that differ in the instruction and data cache are replaced by preloading other blocks that will be needed in the future. Finally, a safe estimation of the WCET of the modified program can be made.

The integration of the program modification and our WCET estimation method described here is the one used in the next section where we evaluate the program modification method and also compare it with the pessimistic serial-execution method.

5 Experimental Evaluation

We have evaluated the amount of pessimism introduced when estimating the WCET of seven benchmark programs, using the two different methods presented in Section 4, the pessimistic serial-execution method, and the program modification method. The modeled architecture is the one presented in Section 2.2, consisting of a multiple-issue pipeline with instruction and data caches.

The key question to answer is how much pessimism is introduced by the two methods. If the pessimism is too severe, it will prompt for advancements in timing analysis methods for dynamically scheduled processors. If it is reasonable, previous methods can be used in combination with the method presented in this paper to enable tight estimations of WCET for programs on dynamically scheduled processors. These are the objectives of the evaluation in this section.

5.1 Methodology

An overview of the seven benchmark programs can be seen in Table 1. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *DES*, *jfdctint*, and *compress*. The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the benchmarks. No optimization was enabled.

To estimate the WCET of the benchmark programs, the WCET simulator and method described in Section 4.3 has been used. The implementation is built upon the instruction-level simulator, PSIM [1], which simulates the POWERPC instruction set. The original simulator has been extended with a WCET algorithm that uses the simulator to estimate the WCET by exploring and merging paths in the program.

The timing model used in the WCET simulator is based on the model of the POWERPC architecture discussed in Section 2.2 with the timing parameters according to

Name	Description
matmult	Multiplies 2 10x10 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
DES	Encrypts 64-bit data
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

Table 1: The benchmark programs used.

Figure 1. However, instead of a detailed simulation model of the pipeline, we use an analytical approach. During simulation, the latency of the simulated instructions is summed together with instruction and data cache miss penalties. This we call the serial time, T_{serial}. We then assume that the time T to execute the program on the dual-issue architecture is:

$$T = \frac{T_{serial}}{2}$$

The relation between T and T_{serial} is obviously not this simple in reality. For a dual-issue pipeline executing only instructions with latency equal to 1 with no cache misses, the above formula would represent the ideal situation of dispatching 2 instructions each cycle. This is often not possible in reality and is highly program dependent. Nevertheless, this formula makes it easy to compare the different estimation methods. When estimating the WCET our model automatically produces the pessimistic serial-execution estimate, which then always over-estimates the actual WCET by at least a factor of 2.

When modifying the programs we used `sync` instructions to handle the pipeline state and preload instructions to handle the instruction and data cache states as described in Section 4.3. We assumed that a single `sync` placed at a merge point in the program incurs a penalty of 5 cycles in the dual-issue architecture. When one `sync` instruction are placed before and one after a variable-latency instruction, we assumed a penalty of 8 cycles, i.e., the second `sync` incurs less penalty than the first one since the pipeline is already flushed by the first `sync`. When adding preload instructions, the program becomes larger. The effect of this on the latency and possible additional instruction cache misses has been estimated manually and accounted for in the results. Three integer multiply instructions were assumed to be variable-latency: `mulhw`, `mulhwu`, and `mullw`. The multiply immediate instruction, `mulli`, and all other instructions were assumed to have fixed latencies.

Program	Actual WCET	Estimated WCET: Serial Method WCET	Serial Method Ratio	Unsafe program WCET	Unsafe program Ratio	Modified program WCET	Modified program Ratio	Modified slowdown
matmult	5283287	10566574	2	5283287	1	6323287	1.20	1.20
bsort	230490	460981	2	230490	1	256854	1.11	1.11
isort	2085	4170	2	2085	1	2325	1.12	1.12
fib	797	1594	2	797	1	797	1	1
DES	186166	372716	2.002	186358	1.001	186358	1.001	1
jfdctint	9409	18819	2	9409	1	9921	1.05	1.05
compress	16486	109167	6.62	54583	3.31	69291	4.20	1.27

Table 2: The estimated WCET using the serial method.

5.2 Evaluation results

The results from our evaluation of the seven benchmark programs can be seen in Table 2. The actual WCET has been determined by simulating the program using the worst-case input data, or using random input data if the worst-case input were to complex to determine. The table also shows the estimated WCET when using the serial method and when using the modified program. Also included for comparison purposes is the unsafe program estimate, i.e., the dual-issue timing model has been assumed but no program modifications have been made. This is unsafe since timing anomalies can lead to an underestimation of the WCET. The ratio columns in the table is the estimated WCET values divided by the actual WCET. The modified slowdown is the modified program estimate divided by the unsafe program estimate and shows the amount of pessimism introduced when modifying the programs.

The serial method overestimates the WCET by at least a factor of 2. This is expected and is a result of our assumed timing model. However, for *DES* and *compress*, additional sources contribute. In *DES*, the small additional overestimation is due to data accesses with an unknown reference address. These unpredictable accesses must not be cached in order to keep the cache state predictable. This is accomplished by mapping the accessed data structures into a non-cacheable part of the memory. Then, unpredictable accesses will not interfere with the cache and will always cause a cache miss. In *compress*, a small part of the overestimation is also due to unpredictable data accesses. In addition to this, the path analysis fails to eliminate all infeasible paths due to a pessimistic upper bound on a loop (a more thorough description of this loop can be found in [6]).

The estimated WCET using the modified programs is lower than the serial estimate for all examined programs. In *fib* and *DES*, the program modifications method gave no slowdown at all since no modifications were needed. These two programs contain no variable-latency instructions and during the analysis, no merging was done.

In *matmult* and *jfdctint*, the slowdown is caused entirely by variable-latency instruc-

tions. No merging was done during the analysis. In *jfdctint*, variable-latency multiplications are only used in the beginning of the program and the inserted `sync` instructions have therefore quite small impact on the estimated value. In *matmult*, however, the multiplications are common and the inserted `sync` instructions give a slowdown of 20 %.

For the remaining programs, *bsort*, *isort*, and *compress*, it is the merging that contribute most to the slowdown. In *bsort* and *compress*, there are a small number of variable-latency multiplications but the effect of those instructions is negligible. In *bsort* and *isort*, the merging occurred at one place in the program. At this place, a `sync` instruction were added, which resulted in a slowdown of 11 % and 12 % for *bsort* and *isort*, respectively. The highest slowdown experienced, 27 %, was for *compress*. This is explained by the fact that merging occurred at four different places in *compress*, each requiring a `sync` instruction.

At the merge place in *bsort* and *isort*, and at two of the four merge places in *compress*, preload instructions for the instruction cache were needed. At these merge places, the instruction cache states differed in the paths being merged. The number of blocks to preload varied between 6 and 10 among the three programs. By preloading blocks that were needed along the worst-case path no extra cache misses occurred and the effect of these preload instructions is very small compared to the merging. The data cache states never differed when merging paths in the programs.

In summary, our program modification method can perform well in conjunction with our symbolic execution method for all our benchmark programs. It works especially good for programs that have few variable-latency instructions and only one feasible path so that merging is avoided when analyzing the program. On the other hand, if a program contains many variable-latency instructions or many feasible paths then the serial method could perform nearly as well or maybe better. For example, if optimization is enabled when compiling *matmult*, the variable-latency multiplications becomes relatively more frequent. This would change the slowdown from a factor of 1.2 to approximately 1.5, closing in on the serial method.

6 Discussion and Future Work

The results show that our program modification method can be used to obtain safe and fairly tight estimations of the WCET for our benchmark programs. This suggests that for a certain class of programs, running on dynamically scheduled processors, it is possible to make safe and tight estimations of the WCET. However, to use the method there must be some support in the architecture to be able to explicitly control the state of caches and the resource allocation in the pipeline. Ideally, one would need explicit program control of all internal state in a processor that may influence the future timing of instructions. If no support exists for explicit control of the state of caches or the

pipeline, then one is forced to use the serial estimation method which often leads to more pessimism in the estimated WCET.

When using the program modification method the resources in the processor can be used out-of-order except at the modification points in the program where we force an in-order execution. An important consequence of this is that we must statically account for all unknown events and modify the program at the proper places. This forbids the use of preemptive scheduling where a program can be interrupted at any time. However, limited preemption would be possible by treating preemption points in the program as being similar to merge points. The cache and pipeline state must be predictable at all points, regardless of the program being preempted or not. The serial method does not rely on making unknown events safe and can be used together with preemptive scheduling.

It is quite possible that an better analysis method can be invented that results in tighter estimations of the WCET. However, when the processor allows out-of-order resource allocation, timing anomalies can occur. A better analysis method could avoid the program modifications, but each unknown event must still be statically known and statically analyzed. An alternative interesting option would be to include the possibility to control the resource allocation in a processor. Then, the processor could be forced to allocate resources in-order resulting in a stable scheduling of instruction but probably also lower performance.

In this paper, we have only dealt with the handling of caches and the basic pipeline. To make the methods presented here useful, other features in an architecture must also be analyzed. For example, further research is needed to analyze the effect of speculative branches and branch history buffers and how to explicitly control the state of these features. Moreover, we only consider dynamic scheduling done inside the processor. To assure a safe estimate, features outside the processor need to taken into account. For example, it may be necessary to consider contention between accesses from the instruction and data cache going to the main memory since these accesses often occur out of program order.

7 Conclusions

Most high-performance processors today use several features that allow out-of-order execution. We have shown that previous methods fail in estimating WCET because they assume that one can rely on worst-case assumptions for local entities such as instructions and basic blocks to estimate the effect on the overall WCET.

In order make available methods useful, we proposed to make program modifications to make unknown instruction latencies predictable. This enables existing methods to estimate the WCET safely. We applied these program modifications to seven benchmark programs and estimated the WCET of these programs using a model of a

dual-issue pipelined processor with instruction and data caches. We found that the pessimism imposed by the program modifications is less than 27 % for the programs in our benchmark. This suggests that for a certain class of programs, useful estimates of the WCET can be obtained for dynamically scheduled processors.

8 Acknowledgment

This research is supported by a grant from Swedish Research Council on Engineering Science (TFR) under contract number 221-96-214.

References

[1] Andrew Cagney. PSIM, a POWERPC simulator. http://sourceware.cygnus.com/psim/.

[2] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[3] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.

[4] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[5] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 334–345, December 1998.

[6] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *To appear in special issue on Timing Validation, the Journal of Real-Time Systems*, November 1999.

[7] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 47–55, June 1997.

[8] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, December 1998.

Paper III

Empirical Bounds on Data Caching in High-Performance Real-Time Systems

Reprinted from

Technical Report 99–4
Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden, April 1999.

Empirical Bounds on Data Caching in High-Performance Real-time Systems

Thomas Lundqvist and Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Email: thomasl@ce.chalmers.se,

Abstract

In this paper we study to what extent hard real-time programs can exploit the performance potential of data caches. Data structures that are accessed by memory instructions whose addresses are input data independent, can be safely cached. Based on a set of non-trivial programs from the SPEC95 benchmark suite, we find that more than 84% of the data accesses are predictable. With static analysis methods, it appears that a high predictable hit rate can be obtained which can result in tighter estimations of the worst-case execution time.

1. Introduction

Many time-critical real-time applications need high-performance microprocessors to meet their performance demands. When developing software for such systems, a main problem is to verify that it meets time constraints specified as deadlines. The industrial practice is to carefully estimate the worst-case execution time through exhaustive measurements which is both time-consuming and error-prone. This has motivated us and other research groups [1, 2, 3, 13] to automate the task of estimating the worst-case execution time. Our goal is to develop methods for integration in a combined compiler/timing analysis tool. This tool not only generates code but provides also estimations of the worst-case execution time of a program.

The task of estimating the worst-case execution time of a program can be formulated as a graph problem. Consider an acyclic[1] control-flow graph representation of the program, where each vertex is a basic block of instructions or an acyclic graph with its associated (worst-case) execution time. The worst-case execution time is then

1. Constraints on loop bounds, for example, must be available to make the control flow graph acyclic.

the longest path from the entry to the exit point. For simple microcontrollers with constant instruction execution times, fairly tight worst-case execution time estimates can be derived this way [10]. For high-performance embedded microprocessors that employ pipelining and caching, however, this methodology can provide very pessimistic estimations. This is because the worst-case instruction execution time can be an order of magnitude longer than the best-case execution time; in the worst-case an instruction can result in two cache misses which can easily stall the processor for tens of cycles. At the same time, cache memories often handle a vast majority of all memory accesses in a single cycle. This has motivated us to develop methods to predict data cache behavior so as to provide tighter bounds on the worst-case execution time.

The challenge is to determine the set of data accesses that are independent of input data. Such memory instructions are allowed to cache data. Recently a few studies have been published on methods to allow for predictable data caching. The basic method in [7] handles all unpredictable data accesses as if they result in two cache misses because in the worst case, a memory instruction will miss and replace data. If the predictable hit rate is lower than 50%, this method will produce more pessimistic results than if data caches were not used. Our approach is instead to let all memory accesses that are not predictable bypass the cache. Support for this exists in most embedded microprocessors. [6] is concerned with the problem of how to model conflicts between predictable data cache accesses and does not address the problem of how to handle unpredictable data accesses.

Ultimately, the effectiveness of data caches in hard real-time systems is dictated by the fraction of all memory accesses that can be predicted, i.e., are input-data independent. The purpose of this paper is to make an estimation of this fraction by (1) formulating what predictable data caching is and by (2) providing empirical data on what fraction of data accesses that are predictable. We do this by analyzing a set of non-trivial programs from the SPEC95 benchmark suite. While our study is preliminary, our empirical data so far look promising; more than 84% of the data accesses are indeed predictable. This suggests that data caching is effective in hard real-time systems although the challenge is to find tractable methods to come close to this bound. The fraction of such predictable data accesses that can be covered by such methods establishes a practical bound on the effectiveness. In Section 2, we provide our approach to predictable data caching. Sections 3 and 4 present the experimental results and, finally, Section 5 discusses our ongoing work.

2. Predictable Data Caching

Predicting cache behavior can typically be divided into two steps. The first step is to find out the reference address/addresses of all instructions. The next step is to use this information to statically model cache behavior so as to predict its impact on the worst-case execution time. Such a method exists for instruction caches [4].

Recently, some methods to analyze data caches have been proposed [6, 7]. Their approach is to extend older methods that handle instruction caches to also handle data caches. This works because in both cases the goal is to statically determine the set of memory accesses for each execution path in the program. However, one big difference is that the addresses generated from load/store instructions to access the data cache are not always known or predictable. A typical case is when the reference address of a load instruction depends on unknown input data. If the address of all accesses to the data cache are known, it is possible to predict whether each memory access will hit or miss in the data cache. Unknown reference addresses of a memory instruction, however, lead to the pessimistic assumption that the memory instruction will miss in cache and results in poor estimations of the worst-case execution time.

Definition (*Predictable memory instruction*): With a predictable memory access instruction (load/store instruction) we mean an instruction that generates the same reference address or the same sequence of addresses, regardless of the unknown input data, and also regardless of the path taken through the program as long as the path includes the instruction.[2]

Predictable memory instructions will put an upper-bound on the prediction of the number of cache hits. Unfortunately, all predictable memory instructions cannot be determined in practice through static compiler analysis (e.g. dataflow analysis). We will call the set of predictable memory instructions that at the same time can be analyzed as *analyzable memory instructions*. The fraction of all executed memory instructions that at the same time are analyzable will put a practical upper bound on the effectiveness of data caching in hard real-time systems.

An interesting question is how good data cache prediction we can get. How many of the memory accesses have an unknown address? If we succeed to analyze all predictable accesses, what improvement in the form of increased hit rate will we see?

To answer the question of how many accesses are predictable, we decided to take a look at memory accesses from non-trivial programs and we chose the SPEC95 benchmark suite as a suitable object of study. These programs are not written specifically for real-time systems, but are interesting due to two facts. First, they are quite big and represent real programs doing some useful work. Second, they are well-known in the computer architecture community in performance evaluations and thereby suitable to use as a reference.

Given that we can determine the set of predictable memory instructions, the question is how this could be used to reach a high and predictable cache hit rate. To make

2. This means that for a fixed path through a program with only predictable memory instructions, we will get a completely statically known sequence of reference addresses, regardless of variations in input data that still give us the same fixed path.

this upper bound on predictability more useful, it is important that unpredictable memory instructions do not change the cache state. Fortunately, a hardware mechanism that permits us to do this exists in most embedded microprocessors that use caches. We can choose if we want to cache or not to cache each individual data accesses. As an example, PowerPC 403 GA [8] has a double mapped memory address space. This means that one memory location can be reached from two addresses and we can choose different cacheability for the two addresses, one cache-enabled address and one cache-disabled address. Since an unpredictable and predictable memory instruction may potentially access the same memory location, a write-through write policy must be chosen to avoid inconsistency.

Another solution is if the system supports virtual memory. Then cacheability is usually controlled at page level, and we can easily arrange a similar double mapping scheme in this case. There are two ways to exploit this. We can either put our unpredictable data structure in a special region in memory, marked as uncacheable, or we can control each individual load/store instruction and make it cacheable or non-cacheable. In the latter case, we must make sure the cache is consistent. To control the cacheability at a data structure level, we need to control the mappings done by the linker. For an instruction level control one might use compiler support. Data structure control seems a little bit simpler and we have chosen this level for our experimental classification of memory accesses. It is then useful to define the term *unpredictable data structure*. This is a data structure (part of the memory) that is accessed by at least one unpredictable memory access instruction. Such data structures are not cached.

3. Experimental Results

3.1 Methodology

We have classified data memory accesses and determined the fraction of accesses that goes to unpredictable data structures. To this date we have studied two programs from the SPEC95 suite.

The SPEC95 suite consists of 8 C-programs using integer arithmetic and 12 fortran programs using floating point arithmetic. We have examined two programs, Compress, an integer program, and Swim, a floating-point program. Compress is an in-memory version of the common UNIX utility. Swim calculates shallow water equations. Swim was first translated from fortran to C by using f2c, a fortran to C translator, because the simulator we used didn't support fortran programs[3].

---

3. The fortran libraries use unimplemented operating system calls.

To analyze the data memory accesses we ran our programs on the simulator SimICS [9]. SimICS simulates the SPARC V8 instruction-set and emulates the SunOS 5.x operating system. When simulating a program it delivers a stream of memory accesses to our classification and data cache simulation program.

We classify each memory access as one of the categories in Table 1, depending on which type of data structure the access refers to. We first classify each data structure used in the program by manual inspection of the source code as being predictable or unpredictable. This information is fed to the classification program which registers the number of accesses to each data structure. Since we are only interested in memory accesses originating from the application, memory accesses from library code do not affect the statistics in our simulations.

Table 1: Categories of classification

Type	Explanation
Scalar, stack	A single reference address to something in the stack area
Scalar, data	A single reference address to a local or global symbol
Array, data	An access to an array, referenced with a predictable series of addresses
Unpredictable	The exact reference addresses to this data structure depends on unknown input data

All SPEC95 programs can be run with two different data sets, one reference data set and one test data set. The reference data set is quite time consuming to simulate and the test data set is usually too small for a serious analysis. We have chosen a medium-sized data set for each of the two programs that is not too simple and not too time consuming. In the end we plan to run the programs with the full reference data set.

3.2 Compress

For compress, the results of the classification can be seen in Figure 1. A total of 110 million memory accesses were classified and 84% were found to be predictable.

Compress works by first filling an array with random characters. Then, it compresses and decompresses this array 25 times. The compression method used is a modified Lempel-Ziv (LZW), which finds common substrings and replaces them with a variable size code. The core of the algorithm processes one character at a time and uses it as an index to a hash table. We regard the content of the array (the random characters) as unknown input data, all other parameters are fixed.

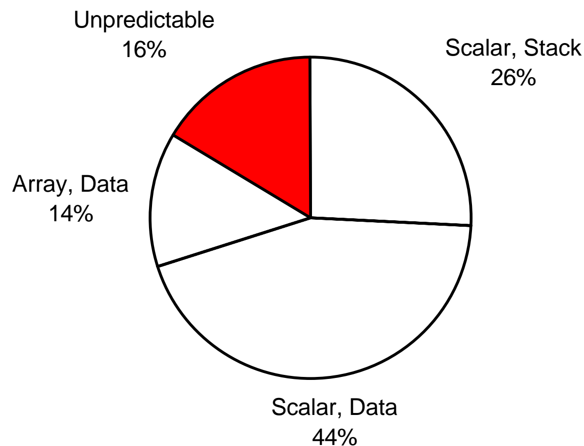


Figure 1: Memory accesses from compress

All stack accesses in compress go to scalar variables (scalar data structures have only one possible reference address) and are therefore predictable[4]. The same is valid for most global and local data. Among the array variables, there are 3 arrays that are unpredictable. These arrays are used as a hash table indexed by the unknown input data.

Most of the predictable array accesses consist of accesses in the form of strides. A simple loop variable is used to index the array. The number of stack accesses is quite high and would be even higher if run on a different processor, due to the fact that the SPARC processor has register windows, which are used for function parameter passing instead of using the stack.

3.3 Swim

For Swim, the results of the classification can be seen in Figure 2. A total of 658 million memory accesses was classified and all accesses were found to be predictable.

Swim consists mainly of matrix calculations. Starting from some initial values, it does the same calculation repeatedly for a fixed number of iterations. The unknown input data in this case would be the initial values of the matrices. The calculations

4. Even if references to the stack area are relative to some stack pointer or frame pointer, it is still possible to statically predict their absolute addresses.

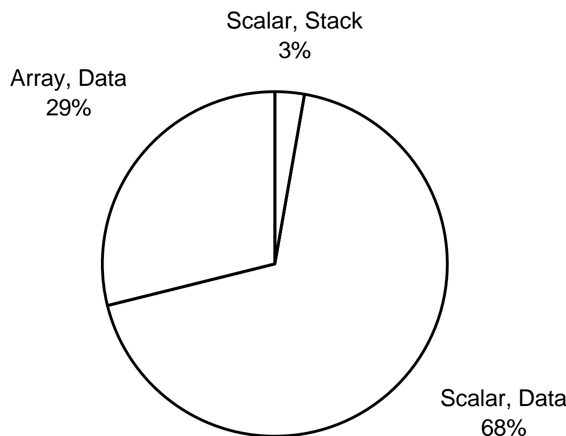


Figure 2: Memory accesses from Swim

done are independent of the actual data inside the matrices. Therefore, all accesses are predictable.

All of the array accesses are in the form of strides. The addresses used is a linear combination of one or two loop index variables. The number of stack accesses is very small. Very few local, stack allocated variables are used, and there is enough registers to hold temporary values during computations.

3.4 Hit Rate Measurements

We have seen that a vast majority of the data memory accesses are predictable. If we cache these predictable accesses, what hit rate would we achieve?

More generally speaking, we would like to know how many of the accesses that are analyzable. This depends on the method used and we will not try to answer that question here, but we can still take a look on how it influences the hit rate.

We have simulated a data cache[5] and let it cache different number of the data structures. For example, we could guess that the scalar variables are analyzable. Then, we cache the references to the scalar variables and do not cache the others (i.e. all other references will count as a cache miss) and see what the resulting hit rate becomes. The result for Compress is seen in Figure 3 and for Swim in Figure 4.

5. The data cache was direct mapped, with a total size of 32768 bytes and a block size of 32 bytes.

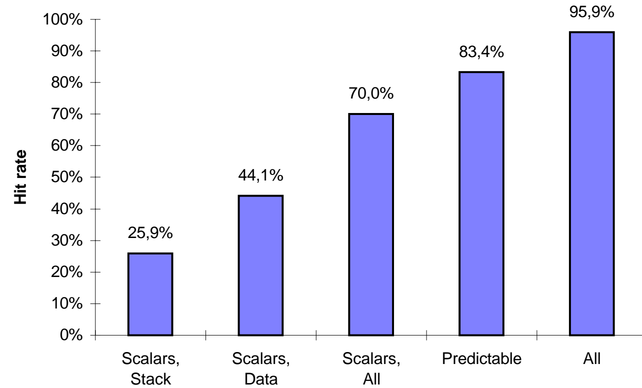


Figure 3: Hit rate when caching different parts of the data accesses for compress

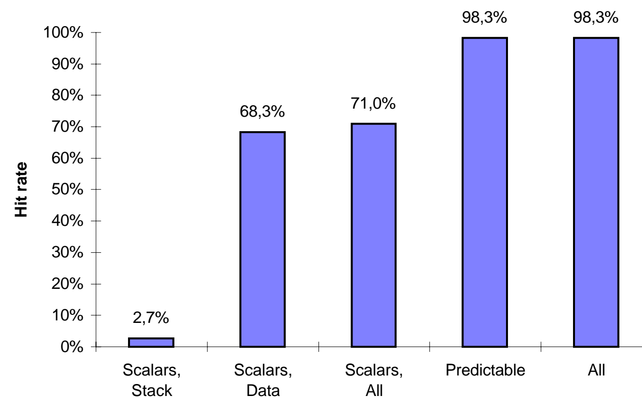


Figure 4: Hit rate when caching different parts of the data accesses for swim

These diagrams tell us that if we succeed in analyzing all predictable accesses, we will reach a fairly high hit rate. In Compress and Swim, the predictable hit rates are 83% and 98% (the same as the actual), respectively. The predictable hit rates are thus close to the actual hit rates.

We can also translate these hit rates into execution times. Let us assume a simple model for a microprocessor with a data cache. Each instruction takes 1 cycle, a cache hit adds no extra cycle and a cache miss will add 10 extra cycles. For our two programs the measured relative frequency of memory instructions is approximately 23%. This simple model gives us the following relation (Figure 5).

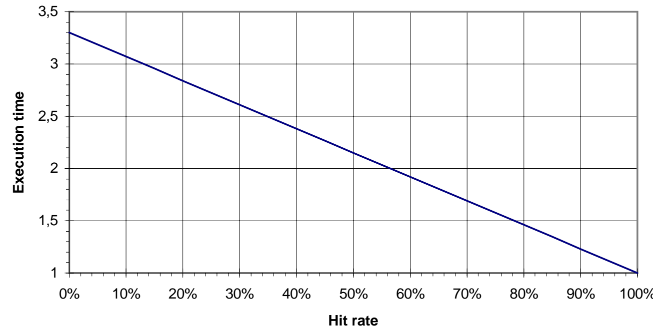


Figure 5: Execution time as a function of hit rate

Figure 5 tells us that if we have a hit rate of 0%, it will take a factor of 3.3 times longer time to execute compared to a hit rate of 100%. For Compress (83%) we get a factor of 1.4. We have thus cut the time by more than half compared to a disabled data cache.

4. Discussion

We have found that the potential of data caching based on the two programs we have studied is high. However, to be able to use the full potential we must find methods to analyze the predictable part of the accesses.

We have implicitly assumed that a program is run without preemption. However, it is possible to use the methodology of classifying accesses as predictable under preemptive scheduling policies as well given that different processes use different partitions of the cache. This can be achieved through software-based cache partitioning as discussed in [5]. This same technique can also be used to allow caching of unpredictable data structures given that they use another partition.

Another interesting question is if we can somehow do better than the predictable limit. The answer to that is yes, we can, but only if we have more information. With more information about the unknown input data (converting it into known input data maybe), we can in some cases make unpredictable accesses into predictable. Also, more information about the program itself (the source code or the algorithm), maybe

in the form of annotations made by a programmer, will also surely prove useful. This and other issues remain open questions for future research.

5. Work in progress

Currently, we are working on a method to estimate the worst-case execution time of a program. A tool using this method is also under construction. Our main design principle is to estimate the worst-case execution time by using architectural simulation. This simulation must be feasible in spite of unknown input data. To accomplish this we have combined architectural simulation with symbolic execution. Thus, the simulator does not only work with real values but also with abstract values.

There are several advantages with doing a functional and temporal simulation of a program. Firstly, we dynamically resolve many dependencies present in the program without the need for extra annotations. This may automatically eliminate many false paths in the program graph. Secondly, we can use more complex annotations, which may be expressed with the help of variables in the program that are known during simulation. Annotations and false-path elimination are means to tighten the estimation on a high level. At the same time an architectural simulator have its strength at a low level since it can accurately mimic the hardware of a system.

Our method is similar to a recently proposed method [11]. However, we do not use a pure branch-and-bound algorithm. Instead we have added a path merge strategy to keep the complexity at a manageable level. The work done in [12] has inspired us, but they are only focusing on the annotation side of the problem.

In the near future we will try to analyze the Compress application and find out how many of the predictable data accesses are analyzable by our method. Our tool will primarily handle a simple architecture with the data cache as the only complex feature.

Acknowledgments

We are indebted to Peter Magnusson of SICS for providing us with the SimICS simulation system. This research is supported by the Swedish Research Council for Engineering Sciences (TFR).

References

[1] D. B. Healy, D. B. Whalley, and M. G. Harmon "Integrating the Timing Analysis of Pipelining and Instruction Caching" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 288-297.

[2] Y Hur et al. "Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 308-319

[3] Y. S. Li, S. Malik, and A. Wolfe "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 198-307

[4] F. Mueller and D. B. Whalley "Fast Instruction Cache Analysis via Static Cache Simulation" in the *Proceedings of the 28th Annual Simulation Symposium*, April 1995, pages 105-114.

[5] F. Mueller "Compiler Support for Software-Based Cache Partitioning" in *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, June 1995.

[6] Y.S. Li, S. Malik, and A. Wolfe "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996, pages 254-263.

[7] S.-K. Kim, S.L. Min, and R. Ha. "Efficient Worst Case Timing Analysis of Data Caching" in the *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, June 1996.

[8] PowerPC 403 GA Users's Manual by IBM. See http://www.ibm.com

[9] The SimICS simulator by SICS. See http://www.sics.se/simics

[10] Allan C Shaw. "Reasoning About Time in Higher-Level Language Software" in the *IEEE Transactions on Software Engineering*, 15(7): 875-889 1989.

[11] Peter Altenbernd. "On the False Path Problem in Hard Real-Time Programs" in the *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, June 1996, pages 102-107.

[12] Andreas Ermedahl and Jan Gustafsson. "Deriving Annotations for Tight Calculation of Execution Time", *EuroPar97*, August 1997 (*to appear*).

[13] Greger Ottosson and Mikael Sjödin. "Worst-Case Execution Time Analysis for Modern Hardware Architectures", *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*.