Processor Pipelines in WCET Analysis

Thesis for obtaining the title of Doctor of Engineering
of the Faculties of Natural Sciences and Technology
of Saarland University.

By
Mohamed Abdel Maksoud

Saarbrücken, April 2015

Date of the colloquium:	24.04.2015
Dean:	Prof. Dr. Markus Bläser
Reporter:	Prof. Dr. Dr. h.c. Reinhard Wilhelm Prof. Dr. Jan Reineke Prof. Dr. Heiko Falk
Chairman of the Examination board:	Prof. Dr. Sebastian Hack
Scientific Assistant:	Dr. Tobias Mömke

Abstract

Due to their nature, hard real-time embedded systems (e.g. flight control systems) must be guaranteed to satisfy their time constraints under all operating conditions. The provision of such guarantee relies on safe and precise estimates of the worst-case execution time (WCET) of tasks. As the execution time depends on both the program and the architecture running it, the growing sophistication of architectures complicates the task of timing analyses. This work studies the impact of the design of the microprocessor's pipeline on the precision and efficiency of WCET analysis.

We study the influence of the design of the load-store unit (LSU) in a modern microprocessor, the PowerPC 7448, on WCET analysis. To this end, we introduce a simplified variant of the existing design of the LSU by reducing its queue sizes. The study contributes empirical evidence supporting the argument that micro-architectural innovations do not improve, and sometimes harm, a processor's worst-case timing behavior.

Building on this evidence, we introduce a compiler optimization to reduce analysis time and memory consumption during the two most-computationally-demanding steps of WCET analysis. With our prototype implementation of the optimization, we observe an analysis speedup of around 635% at the cost of an increase in the WCET bound of 6%. Moreover, under a less precise yet significantly faster variant of the analysis, the WCET bound is decreased by 5% while the analysis is sped up by 350%.

Zusammenfassung

Eingebettete harte Echtzeitsysteme (wie z.B. Flugkontrollsysteme) müssen ihre vorgegebenen Laufzeitgarantien erfüllen. Diese Laufzeitgarantien basieren auf sicheren und präzisen Schranken für die maximale Ausführungszeit (WCET) der Programme. Die Ausführungszeit von Programmen hängt sowohl von dem Programm selbst ab als auch von der Hardware-Plattform, auf der das Programm ausgeführt wird. Die wachsende Komplexität der Hardware-Architekturen erschwert die Berechnung sicherer und präzier Laufzeitschranken (WCET-Analyse). Diese Arbeit untersucht den Einfluss der Pipeline eines Mikroprozessors auf die Präzision und Effizienz einer WCET-Analyse.

Wir untersuchen den Einfluss der Load-Store-Unit (LSU) eines modern Mikroprozessors, des PowerPC 7448, auf eine WCET-Analyse. Wir entwickeln eine vereinfachte Variante der LSU, in der die Warteschlangen verkleinert wurden. Unser Experiment stützt die These, dass mikroarchitektonische Innovationen keinen generellen Fortschritt darstellen, sondern manchmal auch schaden können, wie hier im Beispiel der Bestimmung des Worst-Case-Zeiterhaltens eines Prozessors.

Weiterhin schlagen wir eine Compiler-Optimierung zur Reduzierung der Analysezeit und des Speicherverbrauchs der WCET Analyse vor. Mit unserer Prototyp-Implementierung dieser Optimierung ist eine Reduzierung der Analysezeit von ca. 635% auf Kosten einer 6%-Erhöhung in der WCET-Schranken zu beobachten. Unter einer schnellere Variante der Analyse wird die WCET-Schranke um 5% verringert während die Analyse um 350% beschleunigt werden kann.

Acknowledgements

First, I would like to thank Prof. Reinhard Wilhelm for giving me the opportunity to work under his supervision, for giving me the liberty to pursue the subjects that interested me, and for his enormous support and patience.

I would also like to thank Prof. Jan Reineke: the contributions presented in this thesis owe much of their quality to his insightful input and scrutiny. And not to forget all the football games we played together.

My colleagues in the university have made work much easier and more pleasant: Sebastian Altmeyer, Ilina Bach, Peter Backes, Rosy Faßbender, Sebastian Hahn, Florian Haupenthal, Stefanie Haupert-Betz, Jörg Herter, Michael Jacobs, Ralf Karrenberg, Philipp Lucas, Sandra Neumann, Markus Pister and Marc Schlickling.

Many thanks go to Prof. Sebastian Hack, Prof. Heiko Falk and Dr. Tobias Mömke for being in my thesis committee.

I am grateful for the scholarship I received from the International Max Planck Research School for Computer Science.

I am also grateful for the experience I gained working for AbsInt Angewandte Informatik and for providing the tools which enabled the experiments performed in this work.

And before all, my gratitude goes to my parents for their unconditional dedication, support and encouragement.

Contents

1 Introduction **1**

2 Processor Pipelines in WCET Analysis **5**
2.1 Processor Pipelines 7
2.2 Worst-Case Execution Time Analysis 8
2.3 The Freescale PowerPC 7448 14
2.4 A Quantitative Analysis of the Effects of Various Split Types . . . 17
2.5 Concluding Remarks 22

3 Taming the Hardware **23**
3.1 The Hardware Modification 24
3.2 Experimental Evaluation 24
3.3 Related Work 30
3.4 Concluding Remarks 31

4 A Compiler-Based Approach for Increasing the Efficiency of WCET Analysis **33**
4.1 The Optimization Pass 34
4.2 Experimental Evaluation 41
4.3 Related Work 50
4.4 Concluding Remarks 52

5 Summary **53**
5.1 Summary of Contributions 54
5.2 Conclusions 54

Bibliography **57**

Appendix A Compiler Optimization Results **63**
A.1 Using the Prediction-File-Based ILP Path Analysis 64
A.2 Using the Traditional ILP Path Analysis 69

List of Figures

1.1 Example of two splits encountered during WCET analysis. 3

2.1 The advances in the performance of processor and memory (relative to the first milestone) [HP06]. 6
2.2 Main components of a timing-analysis framework and their interaction. 9
2.3 An example illustrating the differences between traditional ILP-based path analysis and prediction-file-based ILP path analysis. . . 11
2.4 An excerpt of micro-architectural simulation: concrete vs. abstract. 12
2.5 PowerPC 7448 Block Diagram. 14
2.6 Split types and their proportions (logarithmic scale) for selected benchmarks. 16
2.7 Metrics to quantify the effect of a given optimization over a set of benchmarks. 18
2.8 State space size ratio (logarithmic scale) for each split type. 20
2.9 The (inverse-) correlation between pairs of split types. 21

3.1 Number of splits vs. speedup (logarithmic scale). 27
3.2 Underestimation (logarithmic scale) vs. WCET ratio. 27
3.3 Overestimation ratio vs. the number of splits per basic block (logarithmic scale). 29

4.1 The general operation of the optimization pass. 35

4.2 The effect of executing `sync` on merging micro-architectural states. 36
4.3 An example program and its control-flow graph annotated with $R(x)$ and $C(x)$ computation results. 37
4.4 Aggregate program size increase for different values of aggressiveness. 42
4.5 WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis. 44
4.6 WCET bound increase vs. speedup for several aggressiveness values using ILP path analysis. 46
4.7 WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis for three optimizations: opt, $\overline{\text{opt}}$ and rand. 48
4.8 WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis for three instruction memory configurations. 49

List of Tables

2.1 The Mälardalen benchmarks used in experimental evaluations. . . 19

3.1 WCET bounds and performance metrics using prediction-file-based ILP path analysis. 26
3.2 WCET bounds and performance metrics using traditional ILP-based path analysis. 28
3.3 WCET bounds and overestimation induced by the traditional ILP-based path analysis for the full and simplified architectures. 29

4.1 The Mälardalen benchmarks and their optimization statistics at 40% aggressiveness. 40
4.2 WCET bounds and performance metrics for benchmarks using PF-ILP path analysis (aggressiveness=40%). 43
4.3 WCET bounds and performance metrics for benchmarks using ILP path analysis (aggressiveness=40%). 45

CHAPTER

1 Introduction

> I know there's a proverb which says "To err is human," but a human error is nothing to what a computer can do if it tries.
>
> Agatha Christie, Hallowe'en Party

Computers are used in almost every aspect of our lives today. Beside personal computing devices (e.g. laptops, cell phones), we typically deal with computers several times everyday: when listening to music from an MP3-player, using the ATM, washing clothes or warming a meal; there is a computing system which controls the process. Validating the operation of computing systems is therefore crucial to ensure their usability.

The consequences of improper operation of a computing system defines its criticality. A laptop is not considered a critical computing system because if it responds incorrectly or stops responding altogether (i.e. hangs up), no serious repercussions are entailed. On the other hand, a flight control system, an electronic control unit (ECU) in a motor vehicle, and a cardiac pacemaker are critical computing systems since any hazard in their operations could result in severe (and possibly fatal) consequences.

Excluding the possibility of physical damage to the hardware, computers exhibit improper operation because of flaws in their software. These flaws are attributed to either defects in the software functionality (i.e. the system enters an erroneous state), or inadequacy of the system performance (i.e. the system does not respond fast enough with respect to the surrounding physical environment). In this work we focus on the latter class of flaws.

Computer systems which are subject to a time constraint are called real-time systems (RTS). When the constraint is strict (i.e. the system has to meet *all* deadlines), the system is referred to as a hard real-time system. This is opposed to soft real-time systems where it is allowed to miss deadlines every once in a while. The analyses constructed to verify the timing operation of real-time systems (also known as timing analyses) can be broadly divided into two categories: static analyses and dynamic analyses. Static timing analyses verify a program by analyzing its source and the hardware architecture on which it runs at compile time. Dynamic timing analyses, on the other hand, are based on metrics collected by running the program, possibly for multiple times under various conditions (i.e. inputs). Due to the intractability of running a realistic program under all possible inputs, dynamic timing analysis is not suitable for verifying hard real-time systems.

Making certain that a computing system responds with adequate speed requires performing a so-called schedulability analysis, which ensures all tasks (i.e. programs) will meet their set deadlines. Schedulability analysis uses safe and precise estimates of the worst-case execution times (WCET) of tasks. The WCET analysis, which computes these estimates, depends not only on the program being analyzed, but also on the organization of the computer (i.e. the micro-architecture) on which the program runs.

Modern micro-architectures feature numerous mechanisms to increase performance in the common case. One of the main goals these mechanisms attempt to achieve is to circumvent the disparity in speed between two architectural components: processor and memory. Although their performance has been improving over the past three decades, the rate of speed improvement for memory is humble compared to that for processors. This speed disparity motivated building new innovations in the processor to work around stalling when interacting with the memory. Examples include multiple levels of caches and pipelines with dynamic branch predictors and out-of-order execution facilitated by load-store units with multiple registers buffering pending loads

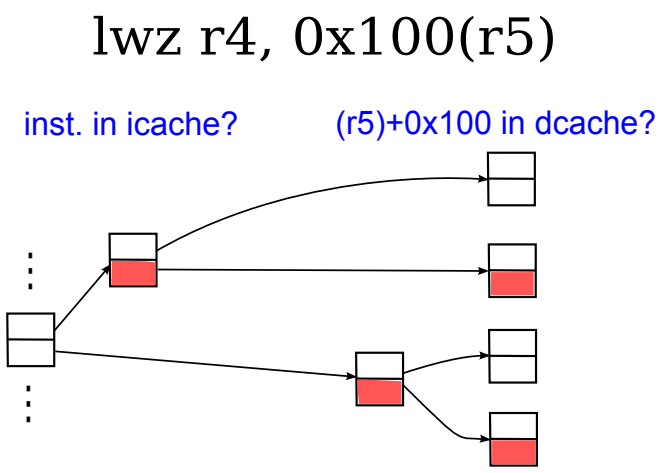


Figure 1.1: Example of two splits encountered during WCET analysis.

and stores. While they proved to be useful for improving system performance in the common case, these innovations complicate the task of analyzing the timing behavior of the system.

For soundness and precision, WCET analysis tools need to determine the possible states of these features throughout the execution of the program being analyzed. However, often, the contents of registers or the cache cannot be precisely determined as they depend on the program's inputs or the particular loop iteration. Whenever the processor's next state depends upon such missing information, the analysis performs a so-called split, accounting for *all* possible successor states. For complex micro-architectures state-of-the-art analyses often track billions of possible micro-architectural states, resulting in long analysis times and high memory consumption.

Figure 1.1 demonstrates splits which can be encountered while performing a WCET analysis. The first split takes place when it is not possible to decide whether the instruction to be fetched is available in the instruction cache. After fetching the instruction, which is a load from a memory address specified by a constant offset from an address stored in a register, another information is possibly missing: is the memory content at this address present in the data cache? In the presence of these two splits, the analysis keeps track of four analysis states for each state at instruction entry. It might be compelling to *only*

keep track of the analysis state which entails longer execution time, since we aim to compute the maximal execution time. However, following this approach potentially results in computing *unsafe* WCET bounds due to the presence of the so-called *timing anomalies* where a locally faster execution leads to a longer execution time of the whole program [LS99; Rei+06].

This work studies the impact of the design of a microprocessor component called the pipeline on the precision and efficiency of WCET analysis. First we simplify a sophisticated component of a modern microprocessor, the PowerPC 7448, and study how this affects the WCET estimates and the analysis efficiency. The simple modification we apply results in a significant speedup of the WCET analysis and surprisingly little or no increase in the WCET bounds. We build on these observations then and introduce a compiler optimization which achieves similar speedups for programs running on an unmodified hardware architecture. This comes at the expense of a slight increase in the WCET estimates. The impact on WCET precision is evaluated based on the more precise (and computationally demanding) variant of the WCET analysis available. There is also a less precise (and much more efficient) variant of the analysis. For the latter variant, our two approaches improve *both* the analysis performance and precision.

Thesis Organization

Chapter 2 presents an overview of the techniques used in modern processor pipelines and its effect on WCET analysis. Then we introduce two approaches (published earlier in [MR12] and [MR14]) aiming to improve the analysis efficiency mainly: one based on modifying the processor's pipeline in Chapter 3 and another based on modifying the program under analysis in Chapter 4. Finally, Chapter 5 summarizes the contributions and insights of this thesis.

CHAPTER

2 Processor Pipelines in WCET Analysis

> Time is like a fast-flowing stream:
> unless you cut it (from source) it
> will shatter you!
>
> Arab proverb

The sophistication of processors has been growing exponentially since the eighties. This growth has been geared towards increasing the processor performance, quantified by measuring their dynamic behavior on executing a set of benchmarks (e.g. SPEC2000 [Hen00]). Given this goal, the outcome has been satisfactory: the processor performance had been growing by approximately 50% from the mid eighties up to 2002 where the growth levels around 20% [HP06].

The growth in processor performance is attributable to new architectural ideas mainly aiming to mitigate the effect of long memory latency. As Figure 2.1 shows, there has been a performance gap between processor and memory. To the end of bridging this gap, architects introduced innovations such as multiple levels of caches and deep pipeline queues such that while expecting a response from the memory for one or more instructions, the processor can potentially work on other instructions.

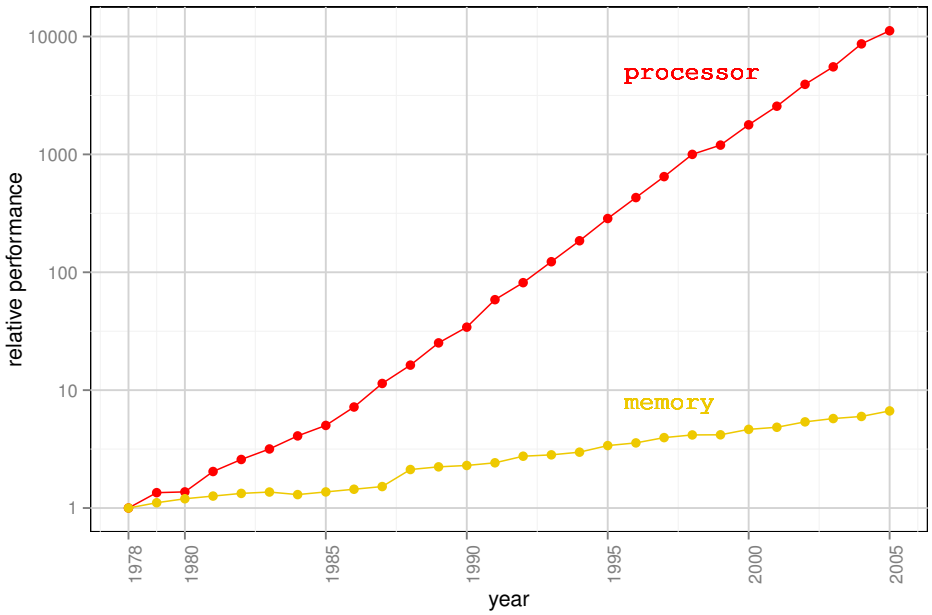


Figure 2.1: The advances in the performance of processor and memory (relative to the first milestone) [HP06].

There is a caveat though: the increased sophistication complicates the task of analyzing the timing behavior of the system. The innovations used to increase the processor performance cause timing analysis tools to explore larger state spaces in order to predict the worst-case performance in a sound manner. This chapter provides an overview on processor pipelines and their role in the context of WCET analysis.

The first section introduces processor pipelines generally. The following section describes a generic WCET analysis framework. Section 2.3 specifies a modern processor with sophisticated mechanisms to increase its performance. The next section presents an empirical study of the effects of various types of uncertainty encountered during analysis.

The terminology used throughout this chapter follows that given in [HP06].

2.1 Processor Pipelines

A microprocessor's *pipeline* refers to the internal processor where arithmetic, logic, branching and data-transfer operations are implemented. The term reflects the fact that the internal processor achieves instruction-level parallelism by means of overlapping the execution of instructions, i.e. by *pipelining* the execution.

In order to achieve greater parallelism (and consequently higher throughput), several techniques are implemented in modern microprocessors:

- *Dynamic scheduling* allows the pipeline to execute instructions as soon as their data dependencies are available, regardless of whether the preceding instructions executed or not (the technique is also called *out-of-order execution*).
- *Branch prediction* enables the pipeline to predict the behavior of branches once they are fetched.
- Branch prediction provides the possibility to execute instructions on the predicted path before the outcome of the branch is computed (i.e. *speculatively*).
- One more technique to increase parallelism is to issue more than one instruction per cycle. The number of issued instructions can be statically fixed as is the case with *Very Long Instruction Word* (or VLIW) processors, or dynamically varying depending on resource availability as is the case with *superscalar* processors.

These techniques have one goal in common: to minimize the stalling in the pipeline for most program executions. To this end, modern microprocessors try to accommodate as many instructions as possible in the pipeline to maximize the likelihood of detecting and exploiting parallelism between them. Accommodating more instructions requires implementing large buffers at various locations in the pipeline. The significant amount of information held by such buffers gives rise to several complications when performing static timing analysis. The next section presents an overview on the worst-case execution time analysis. The complications concerning a particular microprocessor are presented in Section 2.3.

2.2 Worst-Case Execution Time Analysis

The worst-case execution time analysis is a timing analysis which computes an answer to the question: what is the maximal execution time of a program running on a give hardware platform? It is assumed that the program actually terminates to avoid the undecidable halting problem [Tur36]. The assumption is plausible in the context of real-time systems where programs are written in a restricted style. Even with this assumption, computing the exact maximal execution time is intractable for realistic programs and hardware platforms because the analysis has to compute the timing behavior under *all possible inputs*. Therefore, the WCET analysis computes an over-approximation of the maximal execution time instead. The term *WCET* has been used in literature to refer to both the maximal execution time and the computed over-approximation thereof. In this work we shall refer to the computed over-approximation as the *WCET bound*.

The following section presents the different phases of WCET analysis.

WCET Analysis Flow

Over roughly the last decade, a more or less standard architecture for timing-analysis tools has emerged. Figure 2.2 gives a general view of this architecture. The following list presents the individual phases and describes their objectives.

1. *Control-flow reconstruction* [The02a] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered in this phase are dynamically computed control-flow successors, e.g. those stemming from switch statements, function pointers, etc.

2. *Value analysis* [CC77] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. The computed information is used for a precise data-cache analysis and in the subsequent control-flow analysis. Value analysis is the only one to use an abstraction of the processor's arithmetic. A subsequent pipeline analysis can therefore work with a simplified pipeline where the arithmetic units are removed. There, one is not interested in what is computed, but only in how long it will take.

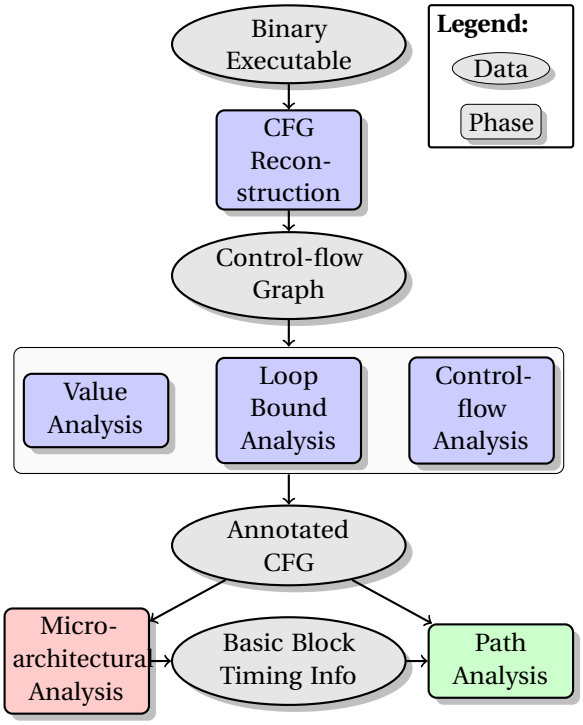


Figure 2.2: Main components of a timing-analysis framework and their interaction.

3. *Loop bound analysis* [EG97; Hea+00] identifies loops in the program and tries to determine bounds on the number of loop iterations; information indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.

4. *Control-flow analysis* [EG97; SM07] narrows down the set of possible paths through the program by eliminating infeasible paths or by determining correlations between the number of executions of different blocks using the results of value analysis. These constraints will tighten the obtained timing bounds.

5. *Micro-architectural analysis* [Eng02; The04; FW99; Cul13] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, combining analyses of the processor's

pipeline, caches, and speculation. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.

6. *Path Analysis* [LM95; The02b] finally determines bounds on the execution times for the whole program by implicit path enumeration using an integer linear program (ILP). Bounds of the execution times of basic blocks are combined to compute longest paths through the program. The control flow is modeled by Kirchhoff's law. Loop bounds and infeasible paths are modeled by additional constraints. The target function weights each basic block with its time bound. A solution of the ILP maximizes the sum of those weights and corresponds to an upper bound on the execution times. In the following, we refer to the kind of path analysis described above as *traditional* ILP-based analysis.

AbsInt's aiT Timing Analyzer

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [FW99; Fer+01; The+03; Hec+03].

The ILP-based path analysis in `aiT` comes in two variants depending on how micro-architectural state graphs are constructed [A3m]:

1. *Traditional ILP-based analysis*, where an ILP is solved to find the worst-case path through the program, given worst-case timings of all basic blocks (possibly in various contexts). In this approach the size of the ILP formulation is independent of the size of the micro-architectural state space. The downside is that the computed WCET bound may be imprecise, because the worst-case timings of consecutive basic blocks may not occur simultaneously on a single architectural path through the program.

2. *Prediction-file-based ILP analysis* (PF-ILP), where a global state graph consisting of micro-architectural states is constructed, and an ILP is solved to find the worst-case path through this state graph. This results

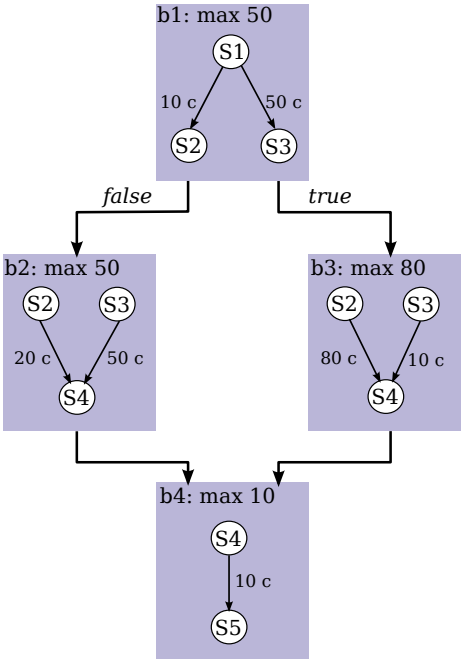


Figure 2.3: An example illustrating the differences between traditional ILP-based path analysis and prediction-file-based ILP path analysis.

in a more precise WCET bound since architecturally-infeasible paths are excluded. However, it comes at the cost of a much larger ILP to be solved, whose size depends on the micro-architectural state space.

To illustrate the difference between the two path-analysis methods, consider the example analysis shown in Figure 2.3. An ILP-based path analysis computes a global WCET bound solely based on the maximum number of execution cycles for each basic block. The WCET is therefore 140 cycles in this case, and the worst-case execution path is `b1`→`b3`→`b4`. However, this result implies an architecturally-infeasible execution trace: `s1`→`s3`|||`s2`→`s4`→`s5`, where trace discontinuity is marked by |||.

On the other hand, the global state graph constructed in a prediction-file-based ILP path analysis excludes such paths and produces a WCET bound of 110 cycles, with the corresponding worst-case execution path: `b1`→`b2`→`b4`, and trace: `s1`→`s3`→`s4`→`s5`.

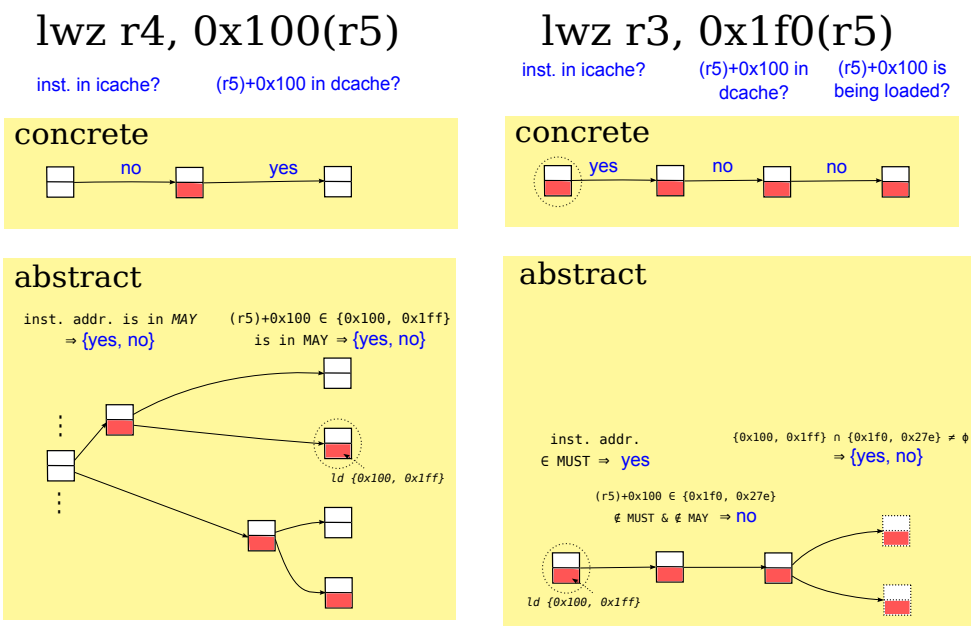


Figure 2.4: An excerpt of micro-architectural simulation: concrete vs. abstract.

Coping with Uncertainty in the Micro-Architectural Analysis

During static analysis, crucial information on program execution such as register and cache contents cannot be determined exactly. When the analysis flow depends on such information, the analysis has to proceed along all possible ways to ensure a sound WCET bound in the presence of timing anomalies [Rei+06]. When the analysis is to proceed in more than one path, the micro-architectural analysis state has to be *split*, increasing the size of the state space and hence reducing analysis efficiency.

Splits induced by unknown cache contents and conditional branch outcomes occur independently of the complexity of the pipeline. Other split types, however, are induced by the missing or partial information about the state of the pipeline.

Figure 2.4 shows an excerpt of micro-architectural simulation in both the concrete case where the state is known precisely and the abstract case where some parts of the state are partially or completely unknown. The microprocessor is assumed to be accommodated with instruction and data caches, and a buffer to store pending memory accesses. The first instruction loads data from a

memory address specified by a fixed offset from the content of a register. To execute the instruction, it has to be fetched from the memory first. The time it takes to fetch an instruction varies depending on whether it is present in the instruction cache or not. In the concrete case, the state of the instruction cache is known and hence there is exactly one possible successor (the one assuming a cache miss in this example). On the other hand, the state of the instruction cache is not known precisely in the abstract case. The cache state is rather available in terms of *must* and *may* information [Rei08]. The instruction address in this example is present only in the *may* set, which requires proceeding along two ways to consider the cache-hit and cache-miss cases. A similar scenario occurs when the load operation is to be executed and we need to check whether the it hits the data cache. The memory address in decidedly present in the data cache in the concrete case whereas it is again in the *may* set in the abstract case. After simulating the first instruction, we end up with four successor analysis states in the abstract case. Note that, in the abstract case, the memory address is represented as an enclosing interval. This reflects the fact that the content of the register `r5` is unknown precisely. This uncertainty and having a buffer of pending accesses causes a different type of splits. On simulating the second instruction starting from an analysis state where the memory access from the first instruction is pending, the analysis has to decide whether the new load request is already being served by the prior request (i.e. if the new request is in the same cache line as the pending one.) The imprecise memory addresses cause one more split in this example.

The split types presented in the example are inherent, i.e. they reflect real cases which do occur on certain executions of the program. There are split types which arise due to the abstraction used. Using the same example, suppose that there is a memory-mapped peripheral accessible at the address `0x17f`. The peripheral has an access latency different than that of the memory. Considering the first instruction in the example, the interval includes the peripheral's address and hence a further split occurs in the cache-miss cases due to multiple access latencies. Such a split is possibly abstraction induced and might be avoided if a different abstraction is used to represent addresses (e.g. by using sets rather than intervals.)

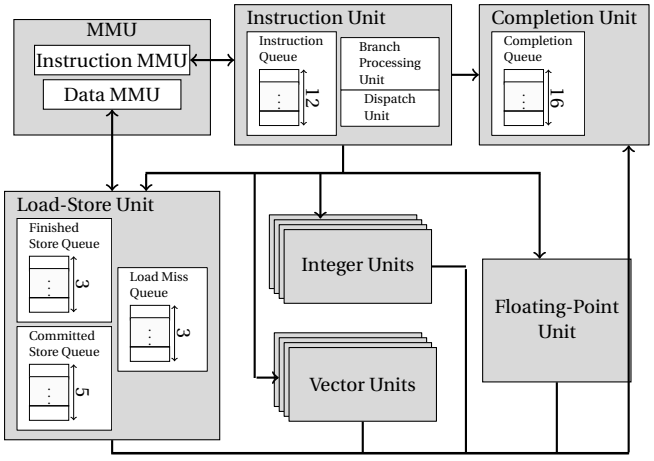


Figure 2.5: PowerPC 7448 Block Diagram.

2.3 The Freescale PowerPC 7448

The PowerPC 7448 is a reduced instruction set computer (RISC) superscalar processor that implements the 32-bit portion of the PowerPC architecture and the SIMD instruction set AltiVec architectural extension. It features a two-level memory hierarchy with separate L1 data and instruction caches (Harvard architecture), a unified L2 cache, four independent integer and four independent vector units for superscalar execution. It also features static and dynamic branch prediction, and a sophisticated load-store unit with long buffers.

"The PowerPC 7448 provides virtual memory support for up to 4 PB (2^{52}) of virtual memory and real memory support for up to 64 GB (2^{36}) of physical memory. It can dispatch and complete three instructions simultaneously" [Mpc]. It consists of the following execution units, depicted in Figure 2.5:

- Instruction Unit (IU): the IU provides centralized control of instruction flow to the execution units. It contains an instruction queue (IQ), a dispatch unit (DU), and a branch processing unit (BPU). The IQ has 12 entries and loads up to 4 instructions from the instruction cache in one cycle. The DU checks register dependencies and the availability of a position in the *completion queue* (described below), and issues or inhibits subsequent instruction dispatching accordingly. The BPU receives branch instructions from the IQ and executes them early in the

pipeline. If a branch has a dependency that has not yet been resolved, the branch path is predicted using either architecture-defined static branch prediction or PowerPC 7448-specific dynamic branch prediction.

- Completion Unit (CU): The CU retires an instruction from the 16-entry completion queue (CQ) when all instructions ahead of it have been completed. The CU coordinates with the IU to ensure that the instructions are retired in program order.

- Integer, Vector, and Floating-Point Units: the PowerPC 7448 provides nine execution units to support the execution of integer, fixed point, and AltiVec instructions.

- Cache/Memory Subsystem: The PowerPC 7448 microprocessor contains two separate 32 KB, eight-way set-associative level 1 (L1) instruction and data caches (Harvard architecture). The caches implement a pseudo least-recently-used (PLRU) replacement policy. In addition, the PowerPC 7448 features a unified 1 MB level 2 (L2) cache.

- Load-Store Unit (LSU): The LSU executes all load and store instructions and provides the data transfer interface between registers and the cache/memory subsystem. The LSU also calculates effective address and aligns data. This unit is described in detail in the following section.

Load-Store Unit The LSU provides all the logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load-store string and load-store multiple operations [Mpc]. The LSU contains a 5-entry load miss queue (LMQ) which maintains the load instructions that missed the L1 cache until they can be serviced. This allows the LSU to process subsequent loads. Unlike loads, stores cannot be executed speculatively: a store instruction is held in the 3-entry finished store queue (FSQ) until the completion unit signals that the store is committed; only then it moves to the 5-entry committed store queue (CSQ). In order to reduce the latency of loads dependent on stores, the LSU implements data forwarding from any entry in the CSQ before the data is actually written to the cache. When a load misses the cache, its address is compared to all entries in the CSQ. On a hit, the data is forwarded from the newest matching entry. If the address is also found in the FSQ, however, the LSU stalls since the newest data at this address could be updated should the store instruction in the FSQ be committed.

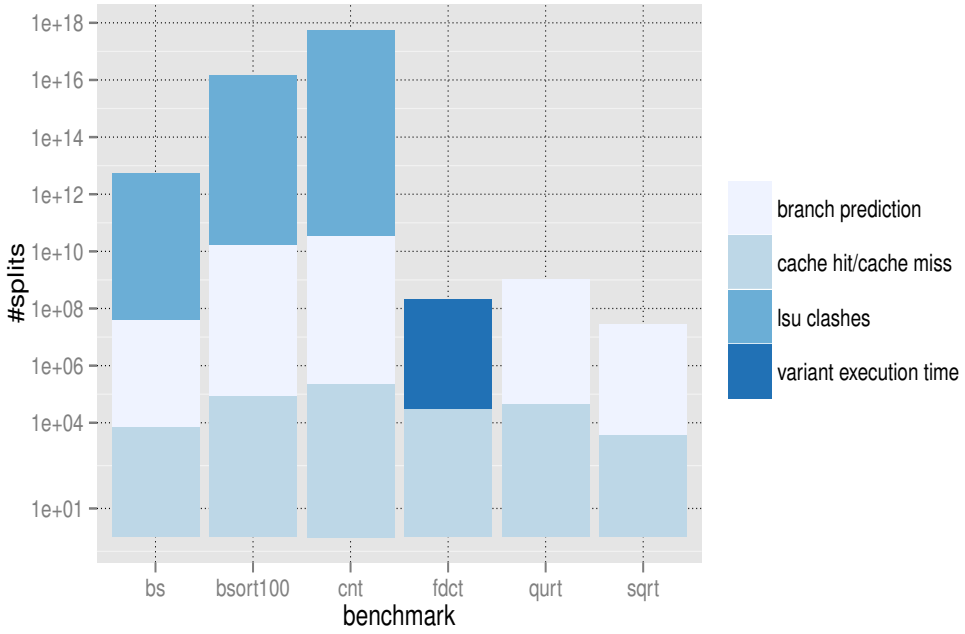


Figure 2.6: Split types and their proportions (logarithmic scale) for selected benchmarks.

Analysis Model of the PowerPC 7448 The various queues in the PowerPC 7448 pipeline necessitate keeping track of a significant amount of information, proportional to the number of instructions the processor can execute concurrently. Due to analysis uncertainty about memory addresses, this translates to a significant amount of splits during micro-architectural analysis.

In the load-store unit, the addresses of different memory accesses are represented by enclosing intervals, rather than exact numbers. As described in the previous section, serving a load that misses the cache involves a number of comparisons to the entries in the store queues. Performing these comparisons on imprecise addresses results in splits whenever it cannot be decided whether two addresses alias or not. The number of comparisons is proportional to the queue occupancies at the time instants when loads that missed the cache arrive.

Another source of splits is branch prediction. Since conditional branches (whose outcome is potentially unknown) are predicted and the instructions are

executed speculatively on the predicted path, a split takes place when deciding whether or not the prediction was correct.

A third source of splits is the execution units featuring variant execution time depending on the operands. For example, a computationally expensive operation such as division takes less number of cycles if the operands have enough leading zeroes.

Figure 2.6 shows the splits encountered on analyzing selected benchmarks and their proportion. Beside cache-induced splits, the load-store unit and branch prediction unit induce a significant proportion of the total number of splits. The effect of the former unit is more pronounced for benchmarks with intensive access to the data memory (i.e. the ones manipulating arrays like `bs`, `bsort100` and `cnt`). In the following chapters we shall see that such benchmarks are the most computationally demanding ones in terms of analysis time and memory consumption.

2.4 A Quantitative Analysis of the Effects of Various Split Types

In the previous section, we have seen that the analysis of the PowerPC 7448 suffers from various types of splits. It is not obvious, though, which of these splits accounts for how much of the state space explored. This section attempts to answer this question empirically. We present a quantitative analysis of the effect of every split type on the size of the state space.

Experimental Setup

In the experiment we analyze benchmarks using a version of AbsInt's WCET analyzer `aiT` for the PowerPC 7448 and construct a micro-architectural state graph for each benchmark. The state graph of a program p and a hardware architecture a encodes all possible micro-architectural state traces resulting from executing p on a. The graph nodes correspond to micro-architectural analysis states. An edge $u \xrightarrow{t} v$ in the state graph denotes a transition from state u to state v which takes t cycles. Splits encountered on analyzing a program are manifested in the nodes with more than one outgoing edge. Every edge is annotated by the choices assumed for the corresponding transition, if any.

$$
\begin{aligned}
\text{ratio}_m(b) &:= \frac{m_{\text{mod.}}(b)}{m_{\neg\text{mod.}}(b)}, \\
\text{ratio}_m^{-1}(b) &:= \frac{m_{\neg\text{mod.}}(b)}{m_{\text{mod.}}(b)}, \\
\text{aggregate-ratio}_m &:= \underset{\forall b \in \mathscr{B}}{\text{geometric-mean}} \{\text{ratio}_m(b)\}, \\
\text{aggregate-ratio}_m^{-1} &:= \underset{\forall b \in \mathscr{B}}{\text{geometric-mean}} \left\{\text{ratio}_m^{-1}(b)\right\}, \\
m\text{-increase} &:= \text{aggregate-ratio}_m - 1, \\
m\text{-reduction} &:= 1 - \text{aggregate-ratio}_m.
\end{aligned}
$$

where

$$
\begin{aligned}
b \in \mathscr{B} &:= \text{the set of all benchmarks,} \\
m \in \mathscr{M} &:= \text{the set of all metrics,} \\
m_{\text{mod.}}(b) &:= \text{is the metric value for the optimized benchmark } b\text{, and} \\
m_{\neg\text{mod.}}(b) &:= \text{is the respective value for the non-optimized benchmark } b.
\end{aligned}
$$

Figure 2.7: Metrics to quantify the effect of a given optimization over a set of benchmarks.

As a metric signifying the complexity of the WCET analysis, we use the number of micro-architectural states explored during analysis (signified as $\#s$). To quantify the effect of each split type on the performance of the WCET analysis, we remove the edges labeled by a certain split type and examine the change in this metric. We combine all split types due to imprecise addresses in the load-store unit under the broad type *LSU clashes*.

We often need to study the effect of a certain modification (e.g. removing edges from the state graph) over a set of benchmarks. We use the definitions listed in Figure 2.7 and benchmarks from the The Mälardalen suite [Gus+10] listed in Table 2.1 throughout this work.

The effect of removing split types are inter-dependent because removing one

Table 2.1: The Mälardalen benchmarks used in experimental evaluations.

Benchmark	Description	# instructions	# bb's	# loops
bsort100	Bubble sort for an 100-integers array.	132	14	2
bs	Binary search for an 15-integers array.	83	11	1
cnt	Counts non-negative numbers in a matrix.	226	17	2
crc	Cyclic redundancy check computation on 40 bytes.	314	29	3
expint	Computes an exponential integral function.	187	25	3
fac	Calculates the factorial function.	61	10	1
fdct	Fast Discrete Cosine Transform.	657	9	2
fibcall	Iterative Fibonacci calculation, calculates fib(30).	54	7	1
janne	Nested loop program.	72	14	2
lcdnum	Read ten values, output half to LCD.	116	27	1
loop3*	Several loop patterns.	160	361	120
ludcmp	Read ten values, output half to LCD.	471	50	11
minmax*	Simple program with infeasible paths.	158	26	0
prime	Calculates whether numbers are prime.	146	23	1
qurt	Root computation of quadratic equations.	234	28	1
sqrt	Square root function implemented by Taylor series.	115	16	1
ud	Calculation of matrixes.	415	39	11

* the benchmark was not found in the official documentation although included in the test-suite distribution.

split type could encompass the effect of removing other split types. To expose this relation, we compute the *independence factor* for each pair of split types (s, t) defined as:

$$\text{independence-factor}(s,t) := \frac{\#s\text{-ratio}_{s,t}^{-1}}{\#s\text{-ratio}_{s}^{-1} \times \#s\text{-ratio}_{t}^{-1}}$$

A high value of this factor implies a weak correlation between the two split types.

Constructing micro-architectural state graphs is technically tedious since the analysis tool-chain does not support the generation of such graphs. We have to reconstruct the graphs out of the graphs of basic blocks. Moreover, such graphs are too large for fairly complex benchmarks to fit in memory. For these reasons, only one Mälardalen benchmark (`fdct`) is used in this experiment beside several small benchmarks written in assembly. The assembly benchmarks attempt to simulate the effect of loading from and storing to imprecise

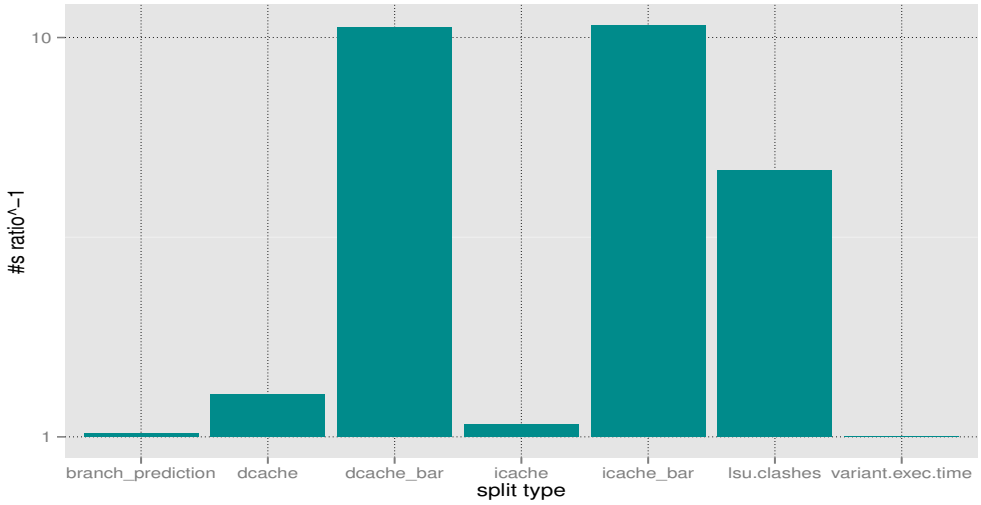


Figure 2.8: State space size ratio (logarithmic scale) for each split type.

addresses, a situation often encountered when analyzing programs with loops that are not completely unrolled.

Results

Figure 2.8 presents the combined state-space reduction computed on removing each split type. Excluding the instruction/data cache miss cases causes the most noticeable reduction. Excluding the load-store-unit clashes also renders a substantial decrease. The effect of removing other split types is negligible. This might be due to the limited selection of benchmarks (e.g. benchmarks which suffer a large number of splits due to branch prediction have too large graphs for this experiment).

The significant effect of removing the cache miss cases could be explained as follows:

- Excluding the data cache miss cases subsumes excluding the LSU clashes.
- Excluding the instruction cache miss cases implies that we examine only the path where the memory subsystem is dedicated to serving the data accesses. Effectively, this shortens the time the load and store instruc-

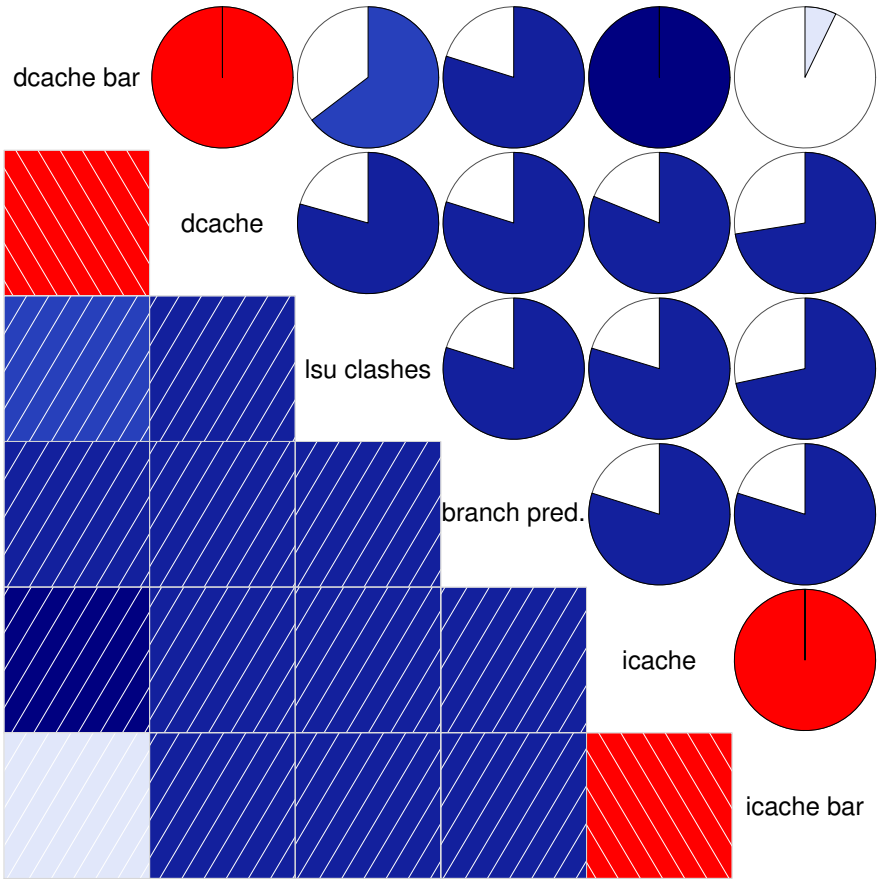


Figure 2.9: The (inverse-) correlation between pairs of split types.

tions reside in the LSU which leads to shorter store queues and ultimately excludes most of the LSU clashes.

This explanation is supported by the correlation analysis shown in Figure 2.9. The graph uses the graphic conventions used in drawing corrgrams as described in [Fri02]. The diagram cells represent the independence factor. In the lower part, the shading color and direction indicates the value of the independence factor. In the upper part, the value is represented as a pie chart. The independence factor for the pairs (icache, $\overline{\text{icache}}$) and (dcache, $\overline{\text{dcache}}$)

cannot be computed because removing both of their components does not allow for a complete simulation of the program. The two pairs are marked with red shading.

Excluding data cache miss cases has a noticeable correlation with excluding the LSU clashes. Excluding instruction cache miss cases is also correlated with excluding the LSU clashes, although not as strongly. The strong correlation between excluding instruction and data cache miss cases is a further evidence they exclude common classes of events.

2.5 Concluding Remarks

Modern hardware architectures cause a great deal of difficulty to the task of timing analysis. It is compelling to utilize the opportunity lying in the middle of this difficulty. It is wished to reduce the state space to be explored during the WCET analysis and still benefit from the performance improvements featured by modern processors. The following chapters present two approaches attempting to mitigate the analysis efficiency problem without degrading the system performance severely. In Chapter 3 we simplify the hardware architecture to make it more analyzable. In Chapter 4 we present a compiler optimization which improves the efficiency of the WCET analysis of programs running on unmodified architecture.

CHAPTER

3 Taming the Hardware

> The gentle overcomes the rigid. The slow overcomes the fast. The weak overcomes the strong. (...) Everyone knows that the yielding overcomes the stiff, and the soft overcomes the hard. Yet no one applies this knowledge.
>
> Lao Tzu, Tao Te Ching

The increasing complexity of today's micro-architectures makes the construction of sound and precise timing models an increasingly time-consuming and error-prone task. Furthermore, the resulting, complex timing models lead to a state-explosion problem in the micro-architectural analysis, drastically increasing overall WCET analysis times.

Most micro-architectural innovations, causing this increase in complexity, like speculation and out-of-order execution, are undertaken to improve average-case performance. In the WCET community it is often argued that many of these innovations do not improve, or even harm, a processor's worst-case timing behavior. There is, however, little hard evidence supporting such claims. This chapter intends to contribute some hard evidence by performing an

empirical evaluation using ABSINT's aiT WCET analysis tool-chain for the PowerPC 7448 processor.

To the end of investigating the influence of the lengths of the Load-Store Unit (LSU) queues on both WCET bounds and WCET analysis times, we introduce a simplified variant of the existing LSU, reducing its queue sizes to a minimum.

The following section describes the modification applied to the LSU. The next section then compares the simplified design with the original design on various benchmarks from the Mälardalen benchmark suite.

3.1 The Hardware Modification

To the end of reducing the number of splits, and thus improving analysis time, we modified the PowerPC 7448 by cutting the queue sizes in the load-store unit. The LMQ, CSQ, and FSQ sizes were reduced to 2, 3, and 2, respectively[1].

We have chosen this modification after several experiments with various other modifications. We tried altering the function units such that their execution times do not depend on the operands. We also tried limiting the speculation level to one rather than three. Out of all modifications we examined, reducing the queue sizes in the load-store unit is the one that produced noteworthy changes both in the precision and efficiency of the WCET analysis.

3.2 Experimental Evaluation

Experimental Setup

The benchmarks were selected from the Mälardalen benchmark suite (cf. Table 2.1). The selected benchmarks are the ones for which the WCET analysis terminated successfully for both architectures.

The aiT analyzer was configured to use traditional ILP-based path analysis (with the CLP solver [Clp]) on all benchmarks and prediction-file based ILP path analysis only on some of them. Although the latter produces more precise WCET bounds, it is more computationally demanding as will be seen in the

[1] This is the strongest simplification we could apply without having to make significant changes to the micro-architectural analysis.

following section, and we were not able to finish the analysis of all of the benchmarks.

We collected the following metrics to quantify the gain in the analysis efficiency and the loss in the predicted WCET bound:

- the time taken by the micro-architectural analysis and the path analysis combined,
- the maximum of the memory consumptions by the micro-architectural analysis and the path analysis.
- the WCET bound,
- the overestimation induced by using the traditional ILP-based path analysis,
- the local WCET bound (lWCET) (computing by pursuing only the micro-architectural states whose execution is *locally* slower) and the amount by which it underestimates the sound WCET bound.

We aggregate the first three metrics obtained for individual benchmarks using the equations in Figure 2.7.

The experiment was performed on a 64-bit AMD Opteron machine with 16 processor cores at 2500 MHz and 64 GB of RAM. As the WCET analysis is not parallelized, we ran multiple analyses concurrently on this machine. As performance metrics, we use the micro-architectural-analysis time and the path-analysis time. On the analyzed benchmarks, these two metrics constitute on the aggregate about 80% and 75% of the whole analysis time for the standard and reduced architectures, respectively.

Results

The analysis results of selected benchmarks using prediction-file-based ILP path analysis are shown in Table 3.1. The unmodified architecture is referred as ¬Mod and the modified one as Mod.

Looking first at the analysis performance metrics, we see that the state space in the reduced architecture is significantly smaller than that of the standard

Table 3.1: WCET bounds and performance metrics using prediction-file-based ILP path analysis.

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Ratio
fac	3,321	3,343	1.007	0.682	0.671	0.984	66.00	66.00	1.000
fibcall	3,346	3,325	0.994	0.526	0.526	1.000	0.00	0.00	1.000
janne	20,005	19,846	0.992	3466.776	2.001	0.001	957.00	101.00	0.106
lcdnum	1,969	1,996	1.014	4.641	1.97	0.424	117.00	91.00	0.778
loop3	39,329	41,199	1.048	5.15	4.183	0.812	135.00	87.00	0.644
minmax	1,629	1,500	0.921	2.762	0.906	0.328	97.00	66.00	0.680
qurt	17,817	17,953	1.008	117.205	24.812	0.212	635.00	224.00	0.353
sqrt	5,096	4,976	0.976	28.528	5.784	0.203	241.00	114.00	0.473
geometric mean			0.994			0.216			0.528

architecture. This is manifested in the consistently lower analysis time and memory consumption, cf. the `janne` benchmark. For simpler benchmarks, such as `fac` and `fibcall`, we do not see significant improvement in the analysis performance. Aggregately, the analysis is sped up by around 460%. Analysis speedup is proportional to how many splits were encountered during analysis. This is demonstrated in Figure 3.1.

Comparing the WCET bounds in both architectures yields a surprise: in half of the cases, the reduced architecture achieves a WCET bound that is *lower* than that of the standard architecture. The aggregate decrease in WCET bound is 0.6%. This decrease could be attributed to the change in the memory access pattern. Alternating the execution of code and data accesses induces less overhead than executing the accesses of each type in chunks. This effect is visible in the benchmarks which do not benefit from the longer queues in the unmodified architecture in terms of performance. To expose this correlation, we consider the local WCET bounds (lWCET) of benchmarks. The lWCET of a given benchmark is less than the sound one if any timing anomaly is encountered. In other words, the more anomalies encountered, the more the lWCET underestimates the sound one. We take the underestimation the lWCET induces as an inverted indicator of how much a benchmark benefits from the sophistication of the unmodified architecture.

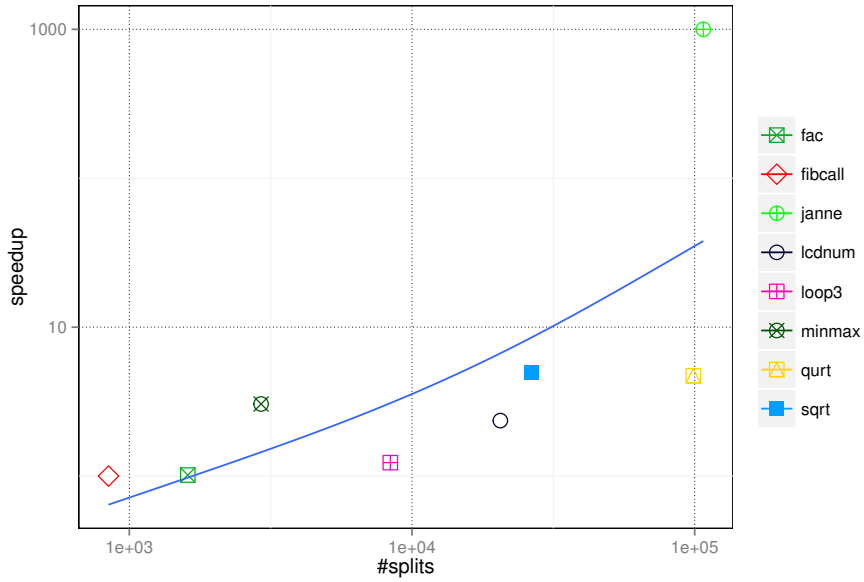


Figure 3.1: Number of splits vs. speedup (logarithmic scale).

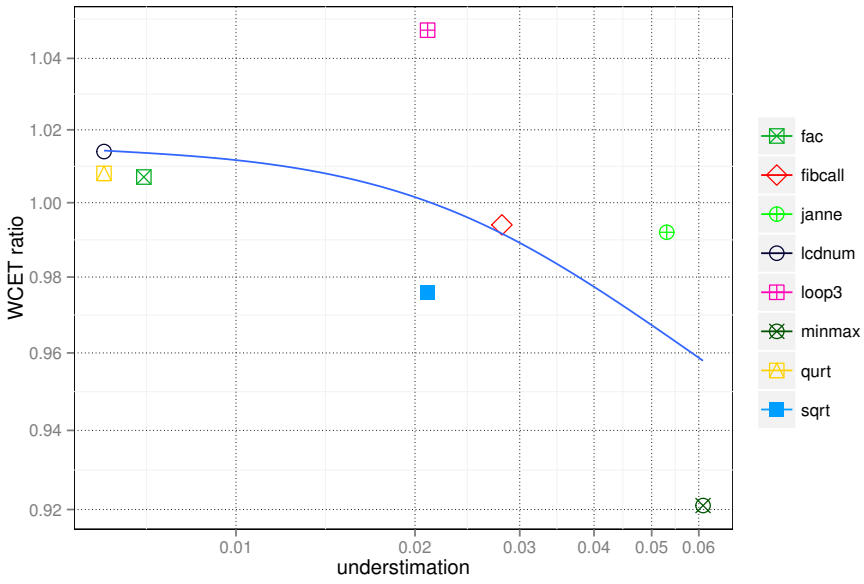


Figure 3.2: Underestimation (logarithmic scale) vs. WCET ratio.

Table 3.2: WCET bounds and performance metrics using traditional ILP-based path analysis.

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Ratio
bs	11,082	9,807	0.885	201.564	15.863	0.079	163.00	74.00	0.454
cnt	44,285	38,399	0.867	16842.557	489.412	0.029	2772.00	235.00	0.085
expint	13,610	13,536	0.995	2.512	2.007	0.799	67.00	67.00	1.000
fac	4,173	4,015	0.962	0.601	0.591	0.983	66.00	66.00	1.000
fibcall	3,685	3,530	0.958	0.49	0.514	1.049	0.00	0.00	1.000
janne	28,172	21,034	0.747	19.864	1.072	0.054	314.00	74.00	0.236
lcdnum	2,538	2,506	0.987	3.035	1.365	0.450	82.00	66.00	0.805
loop3	53,986	53,879	0.998	4.334	3.68	0.849	103.00	87.00	0.845
minmax	1,987	1,898	0.955	1.675	0.821	0.490	66.00	66.00	1.000
qurt	26,363	21,742	0.825	60.58	13.598	0.224	218.00	106.00	0.486
sqrt	7,120	5,576	0.783	14.653	3.267	0.223	114.00	74.00	0.649
geometric mean			0.901			0.284			0.567

Figure 3.2 shows the relation between the WCET bound ratio of the modified and unmodified architectures and the lWCET underestimation on the unmodified architecture. With the exception of one outlier, `loop3`, benchmarks with larger underestimation on the unmodified architecture run consistently faster on the modified one.

Using the less precise, yet significantly more efficient traditional ILP-based path analysis, more benchmarks were analyzed. The analysis results and performance metrics are shown in Table 3.2.

We observe a lower aggregate analysis speedup of around 350%. This is because this variant of path analysis does not benefit from the reduced state space, since it operates at the level of basic blocks.

The WCET bound improvement is more pronounced using this path-analysis variant. This is not surprising since a larger number of paths with different timings through basic blocks, as is the case for the standard architecture, makes it more likely for the path analysis to compute an architecturally infeasible worst-case execution path. Using the example in Figure 2.3, if the first basic block had a single terminal state rather than two, the ILP analysis would compute a bound as precise as the the one computed using the PF-ILP.

Table 3.3: WCET bounds and overestimation induced by the traditional ILP-based path analysis for the full and simplified architectures.

	¬Opt			**Opt**		
Benchmark	PF-ILP	ILP	Ratio	PF-ILP	ILP	Ratio
fac	3,321	4,173	1.257	3,343	4,015	1.201
fibcall	3,346	3,685	1.101	3,325	3,530	1.062
janne	20,005	28,172	1.408	19,846	21,034	1.060
lcdnum	1,969	2,538	1.289	1,996	2,506	1.256
loop3	39,329	53,986	1.373	41,199	53,879	1.308
minmax	1,629	1,987	1.220	1,500	1,898	1.265
qurt	17,817	26,363	1.480	17,953	21,742	1.211
sqrt	5,096	7,120	1.397	4,976	5,576	1.121
overestimation			31.04%			18.20%

We compute the WCET-bound overestimation induced by the ILP path analysis for both architectures in Table 3.3. The overestimation on the simplified architecture is around half of that on the full one. A further investigation of the benchmarks reveals that the ones with a higher number of basic blocks and whose analysis on the unmodified architecture encounters more splits feature better improvement in the overestimation. This is shown in Figure 3.3.

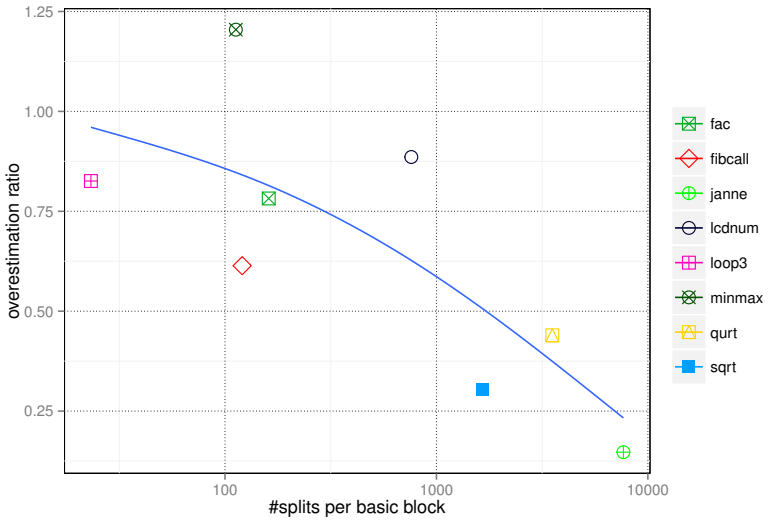


Figure 3.3: Overestimation ratio vs. the number of splits per basic block (logarithmic scale).

3.3 Related Work

While there is an abundance of work proposing more predictable or analyzable micro-architectures, there is not a lot of work that empirically studies the impact of simplifications of micro-architectures on WCET analysis time. Exceptions include the work of Grund et al. [GRG11] and Burguière and Rochange [BR07].

Grund et al. [GRG11] investigate several modifications of the branch target instruction cache of the PowerPC 56x. They observe that using LRU in place of FIFO replacement reduces analysis time drastically, as more memory accesses can be classified as hits or misses, thereby reducing the number of splits.

Burguière and Rochange [BR07] investigate the modeling complexity of various dynamic branch prediction schemes. Here, the modeling complexity is measured by the number of constraints, the number of variables, and the sizes of constraints in an ILP formulation of the behavior of the respective branch prediction schemes. This analysis is based on the assumption that the modeling complexity is strongly-correlated with the resulting analysis complexity. However, the actual analysis times are not analyzed.

Heckmann et al. [Hec+03] focus on the difficulty in modeling various architectural components, including caches and pipelines, and their influence on the precision of the resulting analyses. From their experience in modeling various processors they derive several recommendations regarding the design of processors for real-time systems. Later, Wilhelm et al. [Wil+09] describe properties of memory hierarchies, pipelines, and buses, which make timing analysis more complex and/or reduce its precision. Neither Heckmann et al. nor Wilhelm et al. provide an empirical evaluation of their recommendations.

Approaches aiming at improving predictability or analyzability include the EU projects Predator, Merasa [Ung+10], the PRET project [EL07], and the Java-Optimized Processor JOP [Sch08]. These projects present entirely new processor designs. This makes it difficult to evaluate the impact of individual design choices on WCET analysis times. In the context of the JOP project, Huber et al. [HPS12] analyze the influence of different object cache configurations on worst-case execution time estimates, varying several cache parameters and the background memory. They do not, however, analyze the impact of the design choices on analysis times.

3.4 Concluding Remarks

In this chapter, we have investigated the influence of the design of the load-store unit on WCET analysis, in terms of analysis times and WCET bounds. Reducing the complexity of the LSU results in significantly shorter analysis times, and, surprisingly, sometimes even in slightly lower WCET bounds.

The results indicate that, for the analyzed benchmarks, the sophistication of the load-store unit does not result in a significant increase in the system performance in the worst case. At least not significant enough to justify the hardship it causes when analyzing programs running on this hardware architecture. Simpler and more analyzable architectures can be constructed which render a worst-case performance close to that of the more sophisticated processors.

CHAPTER

4 A Compiler-Based Approach for Increasing the Efficiency of WCET Analysis

> If you want to be a good saddler, saddle the worst horse; for if you can tame one, you can tame all.
>
> Socrates

The hardware optimization introduced in the previous chapter is not one of a kind. The increasing complexity of micro-architectures and its effect on WCET analysis has been observed earlier [TW04] and has lead to a body of work on the design of micro-architectures that aim to reconcile performance with predictability [RS05; Wil+09; Liu+12]. So far, this research has had limited impact on commercially-available micro-architectures.

In this chapter, we explore an alternative approach to new hardware solutions: we propose a compiler optimization that reduces the cost of WCET analysis for complex commercial micro-architectures. This is accomplished by inserting a *synchronization* instruction at selected program points. This instruction stalls the execution until all pending instructions execute to completion, effectively flushing queues in the load-store unit and emptying the pipeline.

This reduces the number of analysis states in two ways. The immediate effect is that many analysis states become *similar* after executing the synchronization

instruction, and hence can be *merged*. In addition, eliminating uncertainty about pending memory accesses in the load-store unit reduces the number of splits on subsequent load-store instructions. The reduced number of analysis states comes at the cost of an increase in execution time due to the stalling induced by the synchronization instruction on the one hand, and the increased program size, which may increase the number of cache misses, on the other hand.

To identify valuable locations to insert synchronization instructions, our optimization estimates, for each program point, the loss in terms of execution time and the gain in analysis efficiency. While the former estimate is based on the loop-nesting level, the latter is computed using annotations obtained by performing a simple static analysis of the program. These annotations provide rough estimates of how long each instruction takes and how many splits it induces.

We have developed a prototype implementation of the optimization for the PowerPC instruction set architecture. We employ a version of AbsInt's WCET analyzer `aiT` for the PowerPC 7448, a high-performance microprocessor used in safety-critical real-time systems, on a set of Mälardalen benchmarks, to evaluate our prototype. Under an expensive prediction-file based path analysis, we observe an analysis speedup of around 635% at the cost of an increase in the WCET bound of 6%. Moreover, under a traditional ILP-based path analysis, the WCET bound is *decreased* by 5% while the analysis is sped-up by 350%.

The following section describe the optimization pass, before we describe the experimental evaluation in Section 4.2. After discussing related work in Section 4.3, we conclude the chapter in Section 4.4.

4.1 The Optimization Pass

High-Level Optimization Approach

The mechanism at our disposal are *synchronization instructions*, described in more detail below, which reduce analysis cost at and after the program point at which they are inserted. As inserting such instructions does not come for free—it increases program size and execution times—blindly inserting synchronization instructions everywhere in the program is not a viable option.

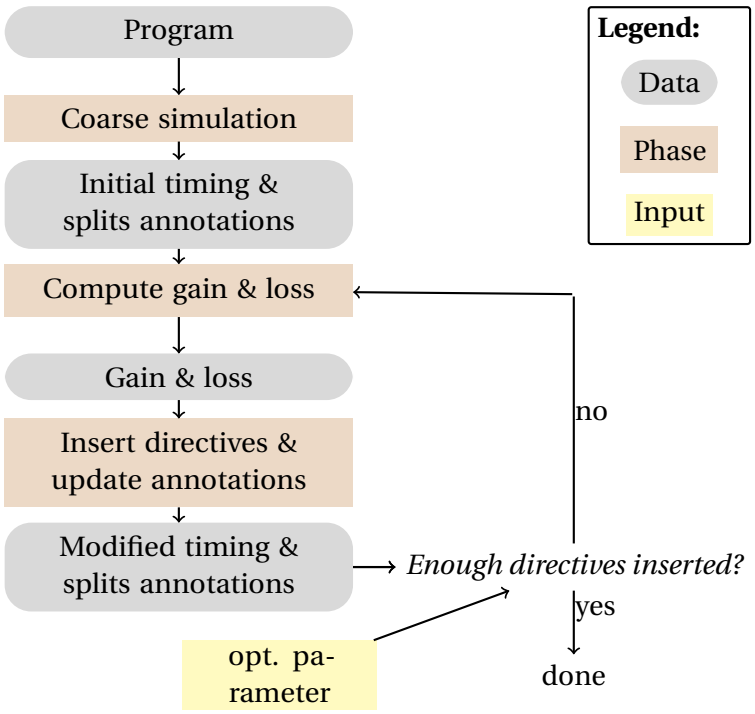


Figure 4.1: The general operation of the optimization pass.

Instead, we follow a simple incremental approach that alternates between the following two steps, until a user-defined threshold is reached:

1. A cheap heuristic is used to estimate both the gain in terms of reduced analysis effort and the loss in terms of increased execution time for each program point.

2. A synchronization instruction is inserted at the program point that maximizes the gain/loss ratio.

Step 1 needs to be repeated in each iteration, because each insertion changes the gains and losses of other program points. Figure 4.1 illustrates this process, whose steps are described in more detail in the following.

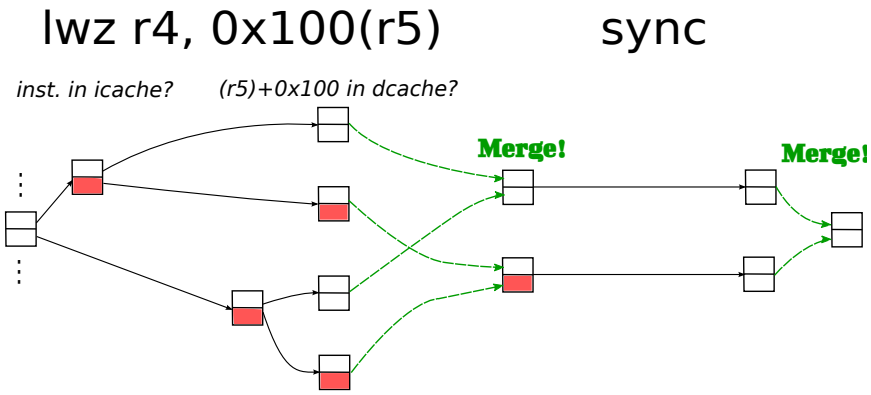


Figure 4.2: The effect of executing `sync` on merging micro-architectural states.

Mechanism: Synchronization Instructions

To eliminate splits due to clashes in the load-store unit, we need to make sure its queues are cleared by the time a new load arrives. The semantics of the data-synchronization instruction (`sync`) is the closest fit for this purpose. Executing `sync` instruction ensures that all preceding instructions execute to completion before any subsequent instruction is initiated [Fre05]. This implies that inserting a `sync` instruction ensures the emptiness of the LSU queues and consequently excludes the possibility of comparing imprecise addresses. Another benefit is that executing the `sync` instruction makes micro-architectural analysis states *similar*. This fosters merging states and hence reduces the number of subsequent states to be explored. Figure 4.2 shows a conceptual demonstration of this effect.

The downside is that inserting `syncs` increases the program size. A longer program will most likely feature a longer execution time and a higher number of splits induced by querying the instruction cache. Moreover, executing the `sync` instruction causes a stalling in the pipeline until all pending operations are completed. This prohibits executing the instructions that follow concurrently and hence prolongs the execution time.

We use the term *normalization point* to refer to a program point before which inserting a `sync` could enhance the analysis efficiency. Normalization points

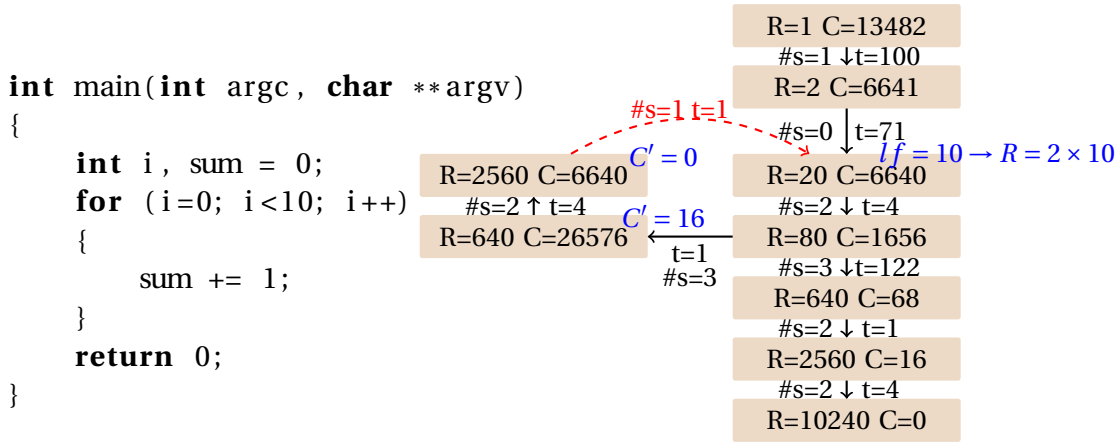


Figure 4.3: An example program and its control-flow graph annotated with $R(x)$ and $C(x)$ computation results.

are the points where one or more splits occur. With this choice, every normalization point corresponds to one or more split types. The normalization point is said to be *realized* if a `sync` is actually inserted before it.

Evaluating Normalization Points

Realizing a normalization point induces gain in terms of the analysis states spared and loss in terms of the increase in the execution time and hence the WCET bound. The additional execution time induced by adding one instruction to a basic block is proportional to how frequently the basic block is executed. This can be approximated as follows:

$$loss = B^n$$

where n is its loop-nesting level and B is a constant signifying the average number of loop iterations. A better approximation of the loss could be computed given information on the loop bounds.

We estimate the number of analysis states spared by realizing a normalization point x as follows:

$$gain(x) = (R(x) - 1) \times C(x)$$

where:

$$\begin{aligned} R(x) &:= \text{the number of analysis states reaching } x, \\ C(x) &:= \text{the cumulative number of states explored per} \\ &\quad\ \text{initial state from } x \text{ to the end of the program.} \end{aligned}$$

To compute $R(x)$ and $C(x)$ for each normalization point, we construct a control-flow graph. Every edge in the graph is annotated by estimates of the time (in cycles) taken to execute the source node (t) and the number of splits encountered during the execution ($\#s$). These estimates are computed by performing a coarse simulation of the program. The simulation keeps track of an approximation of microprocessor state. The state and its evolution are modeled based on the instruction timings listed in the processor manual. To simplify and speed up the simulation, every type of non-determinism in the architecture is avoided by choosing the locally-worst option: memory accesses are assumed to always miss and stores are executed at the lowest speed. Features like branch prediction and speculative execution are not modeled. The only type of non-determinism considered is the one introduced by conditional branches. For such branches, the pass proceeds over all possible paths in a depth-first manner. To handle loops, every basic block is processed exactly once per call location.

Furthermore, we make the control-flow graph acyclic to allow for quick estimation of the two metrics. This is accomplished by finding the feedback edges and removing them from the graph. Finally, program points where no splits take place are discarded unless their removal would affect program structure (e.g. return points).

Computing $R(x)$ proceeds from the program entry point (where $R = 1$) in forward breadth-first manner. Every single split doubles the number of reaching state. To account for loops, we multiply the number of states reaching the *loop head* by an arbitrary constant (we assume that every iteration introduces an additional state). A loop head is the program point to which a feedback edge returns. To formalize, $R(x)$ is computed as follows:

$$R(x) = lf(x) \times \sum \left\{ R(p) \times 2^{\#\mathrm{s}(p,x)} : p \in predecessors(x) \right\}$$

where the loop factor lf is defined as:

$$lf(x) = B^{\left|\{e : e \in \text{feedback-edges} \wedge target(e) = x\}\right|}.$$

Similarly, $C(x)$ is computed from the program end point (where $C = 0$) in a backward breadth-first manner according to the following equation:

$$C(x) = \sum \left\{ 2^{\#\mathrm{s}(x,s)} \times \big(t(x,s) + C(s)\big) : s \in successors(x) \right\}.$$

Removing feedback-edges leaves the last program point in a loop with no successors, we call such points *loop tails*. We initially set $C(x)$ at loop tails to zero and proceed with the computation according to the equation above. As a post-processing step, we propagate the value of $C(x)$ along feedback edges, then we update $C(x)$ from the loop tail backwards to the program point after the loop condition.

An Example Gain/Loss Computation Figure 4.3 shows an example demonstrating the computation of $R(x)$ and $C(x)$ on a simple program with $B = 10$. The graph contains one feedback edge (dashed), the source point is the loop tail and the target point is the loop head.

$R(x)$ assumes the value 1 at the program entry point. At the third program point (which is the target of the feedback edge), $R(x)$ is computed as 20 rather than 2 since the loop factor of this point is 10 (i.e. $lf(x) = B^1$).

$C(x)$ assumes the value of 0 at the program end point and initially at the loop tail. For demonstration purposes, the initial values of $C(x)$ at the loop points are shown as $C'(x)$ in italic type. After computing $C(x)$ for all program points, its value is propagated along the feedback edge (i.e. $C(x)$ is updated to the value of 6640 at the loop tail), and $C(x)$ is re-computed for the loop nodes. In this example, only the loop tail and its predecessors have their $C(x)$ re-computed.

Putting It All Together

To let the user control the trade-off between execution time and analysis efficiency, we introduce the *aggressiveness* parameter. This parameter determines the proportion of the normalization points that should be realized.

Given a certain aggressiveness value, the optimization pass operates in the following phases:

1. A coarse simulation is performed to compute timing and split information for each program point.

Table 4.1: The Mälardalen benchmarks and their optimization statistics at 40% aggressiveness.

Benchmark	Size increase	Time (s)
bsort100	3.03%	0.078
cnt	3.10%	0.078
crc	2.23%	0.396
expint	2.14%	0.184
fac	4.92%	0.044
fdct	1.98%	1.393
fibcall	1.85%	0.022
janne	4.17%	0.065
ludcmp	2.55%	0.936
prime	2.05%	0.086
qurt	1.71%	0.164
ud	2.17%	0.519

2. An annotated control-flow graph is constructed in the way described in Section 4.1.

3. The gain and loss are computed for each unrealized normalization point. The normalization points are then sorted by the ratio of their gain to their loss and the point with the maximum ratio is realized and has its $\#s$ updated accordingly.

   This phase is repeated until the number of covered points is equal to or exceeds

$$aggressiveness \times \left| normalization\_points \right|.$$

In our implementation of the prototype, we use the GCC compiler [Gcc] (version 4.3.2) as a front-end to compile the C sources to PowerPC assembly. The assembly is then parsed to obtain a control-flow graph. Based on this CFG, the optimization pass described above is implemented to produce an optimized assembly file. The assembly parser, simulator and the optimization pass are implemented in Python [DR11]. Finally we use the GCC compiler to assemble the binary executable from the optimized assembly source.

4.2 Experimental Evaluation

Experimental Setup

We used the compiler optimization pass on benchmarks from the Mälardalen suite (cf. Table 2.1) and performed WCET analysis on them using a version of the `aiT` analyzer for the PowerPC 7448. We collected the following metrics to quantify the increase in program size, the gain in the analysis efficiency and the loss in the predicted WCET bound:

- the program size,
- the WCET bound,
- the time taken by the optimization pass, the micro-architectural analysis, and the path analysis combined,
- the maximum of the memory consumptions by the micro-architectural analysis and the path analysis.

We aggregate the metrics obtained for individual benchmarks using the equations in Figure 2.7.

For the first set of results, we configured the `aiT` analyzer to use the more precise, yet more expensive prediction-file based ILP path analysis (with the CLP solver [Clp]). For the second set of results, we performed the same analyses using the computationally-cheaper traditional ILP path analysis. To account for the relatively small benchmarks, the instruction cache size was reduced to 1 KB.

In order to investigate the fitness of our method, we derived two additional optimization passes and used them on the first set of results. The two passes are contingent on the one presented in the previous section, which we call *opt*. They differ only in the method used to select the normalization point to be realized in the following way:

- the *rand* pass selects a normalization point randomly, and
- the $\overline{opt}$ pass selects the normalization point which *minimizes* the gain to loss ratio.

The experiment was performed on a virtual machine with 14 64-bit cores, 54 GB RAM with QEMU/KVM on an AMD Opteron 8360 SE with 16 64-bit cores and 64 GB RAM. As the WCET analysis is not parallelized, we ran multiple analyses concurrently on this machine. To prevent paging, the concurrency level was adjusted such that the combined memory consumption by the running analyses never exceeded the physical memory size.

Experimental Results

Optimization Results Table 4.1 describes the benchmarks used in the experiment along with the percentage increase in program size and time taken by the optimization pass with the aggressiveness parameter set to 40%. The time taken by the optimization is negligible in comparison to the time taken by the value analysis or the micro-architectural analysis as we shall see in the following section. The aggregate increase in program size for different values of aggressiveness is plotted in Figure 4.4. As expected, the aggregate increase is proportional to the aggressiveness value.

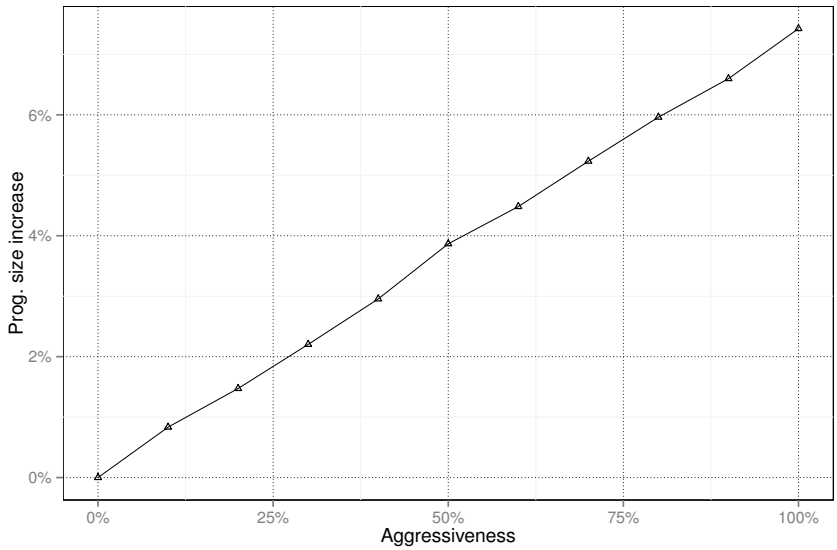


Figure 4.4: Aggregate program size increase for different values of aggressiveness.

Table 4.2: WCET bounds and performance metrics for benchmarks using PF-ILP path analysis (aggressiveness=40%).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	592295	1.0430	13404.45	69.36	193.2367	8186	701	0.0856
cnt	21897	25927	1.1840	88487.60	24.41	3623.1884	21594	95	0.0044
crc	353724	357144	1.0097	24.12	18.54	1.3012	207	179	0.8647
expint	12716	12824	1.0085	2.41	2.67	0.9043	59	66	1.1186
fac	3404	4021	1.1813	0.93	0.46	2.0022	58	0	0.0000
fdct	33637	35158	1.0452	146.50	17.11	8.5626	849	75	0.0883
fibcall	3387	3356	0.9908	0.48	0.39	1.2220	0	0	1.0000
janne	1793	2083	1.1617	0.86	1.1306	0.8845	58	57	0.9828
ludcmp	17750	18986	1.0696	16.99	2.88	5.9061	101	67	0.6634
prime	6801	6911	1.0162	6.25	1.92	3.2551	75	58	0.7733
qurt	18900	18496	0.9786	271.20	69.18	3.9202	822	294	0.3577
ud	14867	15835	1.0651	11.73	3.94	2.9787	83	67	0.8072
geometric mean			**1.0605**			**6.3600**			**0.1440**
geometric mean excl. `cnt`			**1.0500**			**3.5723**			**0.1978**

WCET Analysis Results For each benchmark, the value of the metric is shown for the non-optimized and the optimized versions, along with the ratio between the two values.

First, we consider the metrics collected using the more precise path analysis PF-ILP. The metrics obtained at 40% aggressiveness are presented in Table 4.2. The WCET bound is slightly higher in most of the optimized versions, with an aggregate increase of 6.05%. An increase in the execution time is expected due to the additional `sync` statements that need to be fetched and the stalling in the pipeline it causes. On the other hand, the analysis has a substantial aggregate speedup of approximately 636% and its memory consumption was reduced by about 85%.

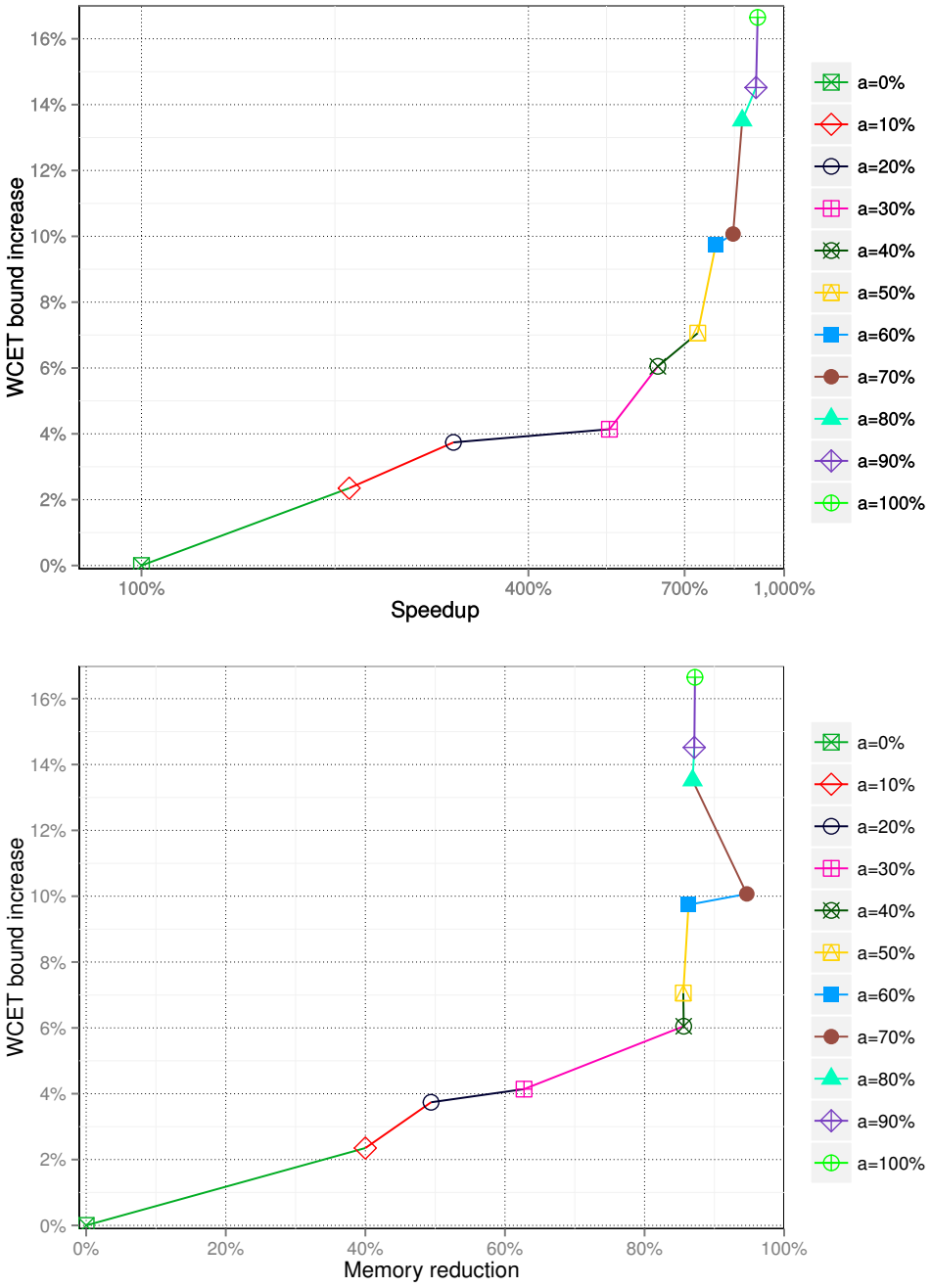


Figure 4.5: WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis.

Table 4.3: WCET bounds and performance metrics for benchmarks using ILP path analysis (aggressiveness=40%).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	767803	1.1226	832.81	40.87	20.3782	6873	702	0.1021
cnt	40422	28654	0.7089	9596.36	17.70	542.2993	3340	90	0.0269
crc	361219	358287	0.9919	15.65	10.34	1.5135	211	178	0.8436
expint	13543	13343	0.9852	1.99	1.51	1.3164	58	58	1.0000
fac	4357	4277	0.9816	0.72	0.42	1.7102	58	0	0.0000
fdct	36459	37420	1.0264	21.86	11.50	1.9010	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.35	1.1780	0	0	1.0000
janne	2313	2339	1.0112	0.60	0.57	1.0615	57	57	1.0000
ludcmp	21945	20798	0.9477	6.16	1.84	3.3476	98	66	0.6735
prime	8354	7435	0.8900	3.58	1.30	2.7629	74	58	0.7838
qurt	27935	23604	0.8450	45.72	14.02	3.2614	266	130	0.4887
ud	18660	18162	0.9733	4.97	2.92	1.6986	82	58	0.7073
geometric mean			**0.9483**			**3.5902**			**0.2048**
geometric mean excl. `cnt`			**0.9737**			**2.2752**			**0.2462**

Surprisingly, some benchmarks, i.e. , `fibcall` and `qurt` show a *decrease* in the WCET bound. This decrease could be attributed to the change in the memory access pattern. Alternating the execution of code and data accesses induces less overhead than executing the accesses of each type in chunks. The benchmark `fac` and `janne` were harmed most by the optimization, with a significant increase in the WCET bound without achieving a proportional speedup. This implies that the gain/loss ratio of one or more normalization points was over-estimated.

The benchmark `cnt` suffers a large increase in the WCET bound too, yet it displays the highest analysis speedup and memory-consumption reduction. Examining the detailed analysis statistics, we found that analyzing the non-optimized version of this benchmark encountered over 16 million splits due to clashes in the load-store unit. The optimization pass with 40% aggressiveness reduced this number to a 40 thousand. This explains the huge gains achieved for this benchmark. Excluding `cnt` from the aggregate values reduces the analysis speedup and memory-consumption reduction while it improves the increase in the WCET bound.

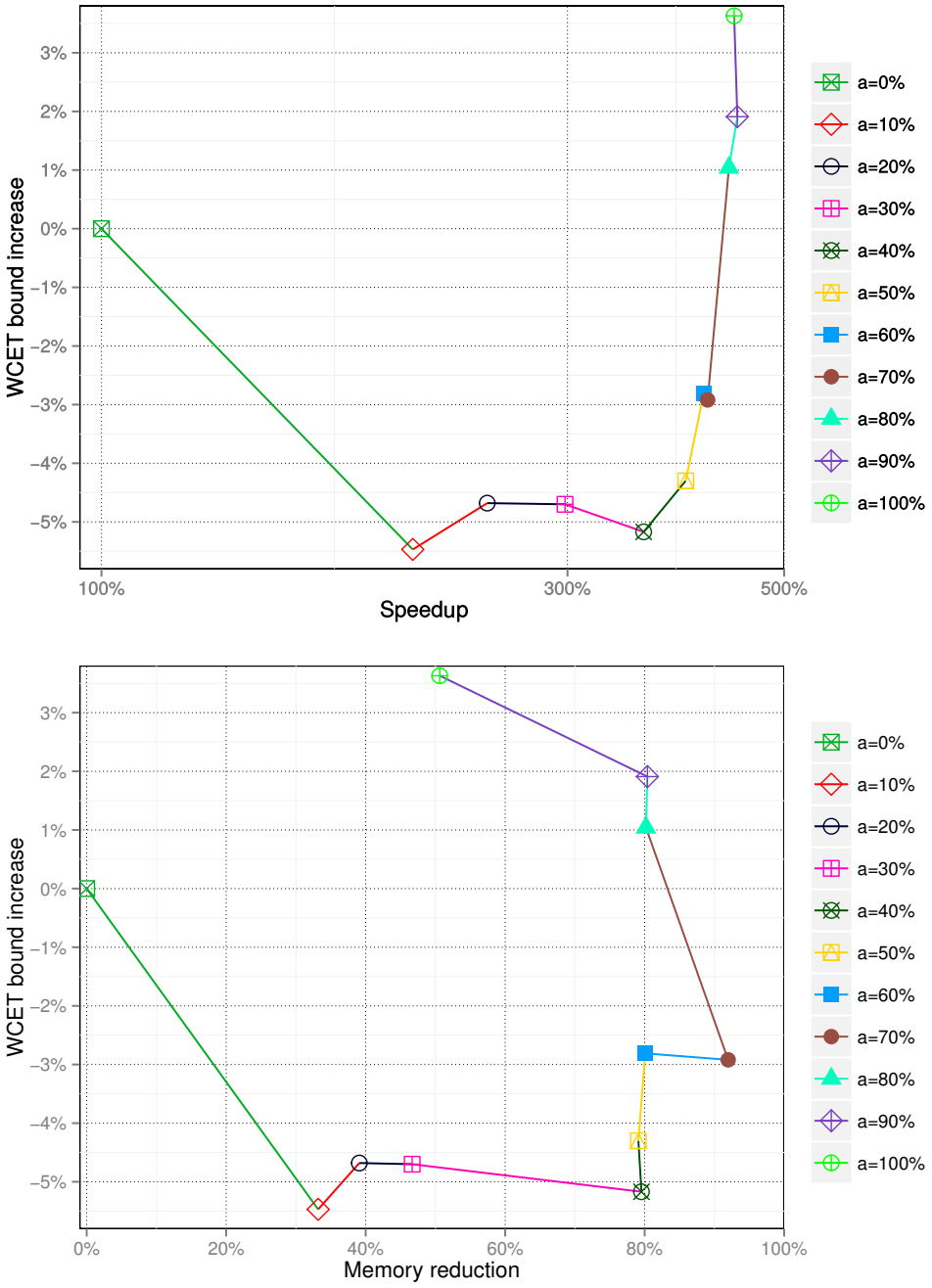


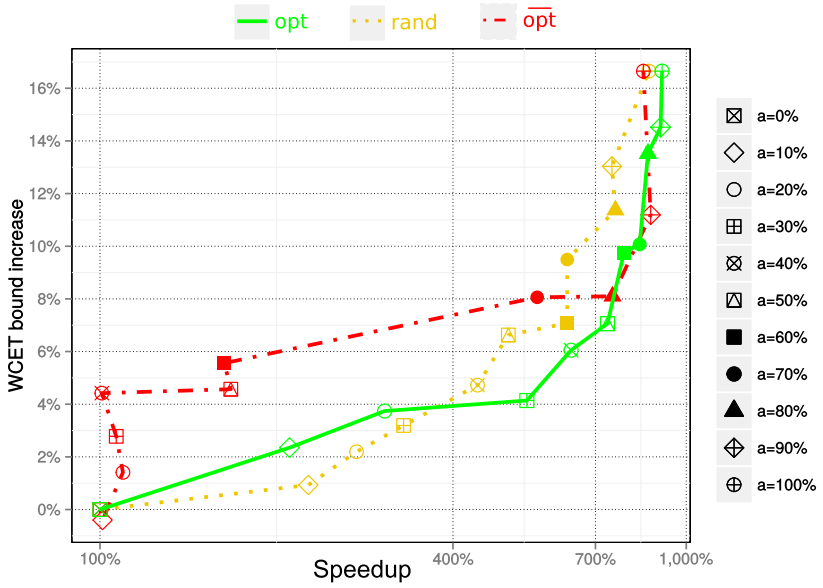
Figure 4.6: WCET bound increase vs. speedup for several aggressiveness values using ILP path analysis.

The second-best speedup and memory-consumption reduction is seen in the benchmark `bsort100`. We can see a pattern here: the benchmarks involving significant number of iterations over large arrays benefit most from the optimization.

To examine the effect of the aggressiveness value, we computed the aggregate WCET bound increase versus the analysis speedup and memory-consumption reduction for several values of the parameter, the results are shown in Figure 4.5. The WCET bound increases proportionally with the aggressiveness peaking around 17%. The speedup also increases proportionally with the aggressiveness up to the value of 90%, peaking close to 900%.

The memory-consumption reduction increases proportionally with the aggressiveness up to 40%, spikes at 70% aggressiveness to around 95% and then declines to around 88% for greater aggressiveness values. A possible explanation of this observation is that there are two factors which affect the memory-consumption reduction: the number of analysis states spared by inserting `sync` instructions and the number of splits induced by the additional queries to the the instruction cache.

Next, we consider the metrics collected using the traditional ILP-based path analysis. The results are shown in Table 4.3 and Figure 4.6. While the speedup and memory-consumption reduction are not as significant as they are in the PF-ILP case (approximately 450% and 90% at maximum, respectively), the aggregate WCET bound increase is consistently *negative* for all aggressiveness values below 70%. The reduced speedup is attributed to the fact that the ILP path analysis does not depend on the complexity of the micro-architectural analysis. The path analysis therefore does not benefit from the reduced size of the state-space in terms of performance. It does benefit though in terms of precision. Realizing a normalization point in a basic block forces the processor to execute all pending operations within the same basic block. Localizing such operations within basic blocks reduces the infeasible combinations of events that the traditional ILP-based path analysis considers to compute the global bound. The noticeable reduction in the WCET bound can therefore be explained by this precision enhancement.



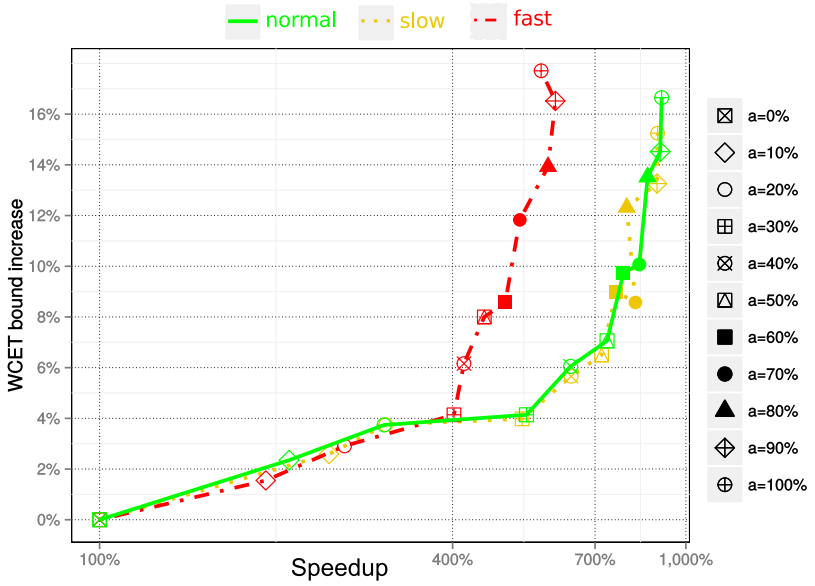
Points are comparable iff they correspond to the same aggressiveness value.

Figure 4.7: WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis for three optimizations: opt, $\overline{\text{opt}}$ and rand.

The Fitness of the Optimization We consider the fitness of our method by comparing the performance of our pass with that of the two contingent passes $\overline{\text{opt}}$ and rand. Figure 4.7 shows the speedup versus the WCET bound increase for the three passes.

The metrics for the rand pass were aggregated over seven independent runs. At the two extreme values of aggressiveness where the selection method is irrelevant, the three passes render identical bound increase and very similar speedup. The slight speedup discrepancy at the full aggressiveness range can be attributed to hazards on the machine caused by other programs running beside the experiment.

The speedup achieved by our pass is greater than that achieved by $\overline{\text{opt}}$ everywhere, and by a big margin for lower aggressiveness values. The speedup is also greater than that achieved by the rand pass except at 10% aggressiveness. These observations indicate that our heuristic function is able to predict the gain correctly to a great extent.



Points are comparable iff they correspond to the same aggressiveness value.

Figure 4.8: WCET bound increase vs. speedup for several aggressiveness values using PF-ILP path analysis for three instruction memory configurations.

The heuristic is not as good at predicting the loss, though. This can be seen by observing that, compared to the rand, the WCET bound increase is always greater (i.e. worse) for our optimization pass. This is not very surprising since the heuristic for computing the loss is rather elementary, and its computation relies upon imprecise estimates of the number of loop iterations. Furthermore, the heuristic does not account for the delay induced by the stalling induced by executing synchronization instructions. This stalling turns out to be a significant component of the increase in the WCET bound as we shall see in the following section.

The Effect of Instruction-Memory Speed One cause of the increase in the WCET bound is the additional synchronization instructions which have to be fetched. The speed of the instruction memory therefore impacts the performance of our optimization. To evaluate this effect, we computed the WCET ratio and analysis speedup with two additional instruction memory configurations: *slow* which assumes a miss penalty of sixteen cycles (this is four

times slower than the *normal* configuration) and *fast* which assumes a uniform instruction memory access penalty of one cycle (i.e. the program is locked in the instruction cache). The outcome is depicted in Figure 4.8.

Executing programs with a slower instruction memory does not seem to affect the analysis speedup. The WCET bound increase is affected though: for all aggressiveness values, the increase is consistently lower (peaking around 15% as opposed to 17% with the normal configuration). Although using a slower instruction memory quadrupled the cost of prefetching additional instructions, it introduced a more dominant benefit: the stalling caused by the semantics of the `sync` instruction does not cause as much performance degradation. The performance degradation induced by executing `sync` instructions is proportional to the amount of instruction-level parallelism lost due to stalling. When the processor waits for longer times for instructions to come from memory, the attainable parallelism (and hence the performance loss) is significantly limited.

Executing programs with a faster instruction memory causes lower analysis speedup. This is attributed to the fact that locking programs in instruction cache excludes one major source of splits: the uncertainty about the state of the instruction cache. This reduces the state-space size and hence reduces the attainable gain in analysis efficiency. The WCET bound increase is slightly lower up to 40% aggressiveness, a value after which the bound increase is consistently higher (except at 60% aggressiveness). A possible explanation is that for lower aggressiveness values, the lowered penalty of fetching `sync` instructions resulted in overall lower WCET bounds. As more synchronization instructions are added, the loss in parallelism induced by executing `sync` instructions outweighs the benefit of fetching them faster.

4.3 Related Work

Recently, significant efforts have been undertaken to develop timing-predictable micro-architectures. The goal of such efforts is to develop micro-architectures that have good worst-case performance and permit sound, precise, and efficient timing analysis.

Wilhelm et al. [Wil+09] recommend using compositional pipelines and separate level-1 caches for code and data with the least-recently-used replacement policy. The recommendation of using the less-sophisticated compositional

pipelines is motivated by the fact that the execution time is often dominated by memory-access times. The optimization we present here is inspired by this observation: we alter the program at specific points to render the stall-on-accident behavior characteristic to compositional architectures. The Java-Optimized Processor [Sch03] by Schoeberl was designed to be WCET friendly. Beside featuring constant instruction execution times, the processor design prevents interleaving instruction fetches with data accesses by loading whole methods on invocation and return into the instruction cache. Rochange et al. [RS05] propose an execution mode of a superscalar microprocessor which excludes interferences between consecutive basic blocks. While this method achieves significant reduction in the analysis complexity, it causes a large slow-down of the system. This approach would roughly correspond to inserting synchronization instructions at the beginning of each basic block. Liu et al. [Liu+12] present the PTARM, a PRET (precision-timed) architecture implementing a subset of the ARM instruction set architecture. The architecture features a thread-interleaved pipeline which exploits thread-level parallelism to combine high throughput with the compositional way instructions are executed within each thread.

Our previous work [MR12] suggests that shortening queues in the load-store-unit of the PowerPC 7448 causes little or no increase in execution times while speeding up WCET analysis significantly. We observe a similar speedup in WCET analysis in the present compiler-based approach as was observed for the hardware modifications suggested in [MR12]. However, in contrast to the present work, the hardware approach did not incur an increase in the WCET bound. The main reason for the difference is that synchronization instructions stall the pipeline.

The compiler optimizations in the WCET domain we are aware of aim at improving the WCET bound rather than WCET analysis efficiency. Falk et al. [FL10] propose a variety of techniques in this direction. Some of the mechanisms used to achieve the reduction are reducing the number of calling contexts for each procedure (and hence improving the precision of the value analysis), implementing a better loop-bound analysis, and reducing the frequency of jumps (and consequently their performance penalty). The first two mechanisms likely reduce analysis time as well. However, these particular optimizations are targeting software rather than hardware aspects of WCET analysis, and are thus orthogonal to the optimization we present in this work.

4.4 Concluding Remarks

We have presented a parameterized compiler optimization pass to increase WCET analysis efficiency. Experimental results confirm that the pass achieves significant analysis speedup at the cost of a small increase in the WCET bound. The optimization also enables the use of traditional ILP-based path analysis with greater precision. Having a parameter to control the aggressiveness of the optimization enables the user to control the trade-off between system performance and analyzability. In contrast to approaches that rely on custom predictable hardware, our compiler-based approach is readily applicable to existing commercial micro-architectures. The optimization pass could use the following improvements:

- by augmenting the optimization with a heuristic model of the stalling induced by executing the synchronization instructions, and
- by using the results of a loop-bound analysis to estimate the increase in execution time due to fetching the synchronization instructions.

Moreover, the results presented in this chapter demonstrate the feasibility of incorporating a normalization flag in the instruction set architecture, should a custom predictable hardware be constructed. This should remove the cost of fetching additional instructions.

CHAPTER

5 Summary

> The more life teaches me,
> the more it shows me my
> intellectual shortage,
> and as I gain more knowledge
> I become more knowledgeable of
> my ignorance.
>
> Abu 'Abdillah al-Shafi'i (767 - 820)

> It is paradoxical, yet true, to say,
> that the more we know, the more
> ignorant we become in the
> absolute sense, for it is only
> through enlightenment that we
> become conscious of our
> limitations.
>
> Nikola Tesla

In this thesis we have investigated the challenges imposed by the sophistication of modern processor pipelines on WCET analysis. The main challenge is the increase in the state space proportional to the processor complexity which ultimately results in a degradation in the performance of the WCET analysis. We have introduced two optimizations to mitigate this problem. The methodology we used to construct and evaluate the optimizations is empirical, mainly because of the intractability of global analyses addressing a relevant problem such as [RS09]. We therefore make no claim of universality for the results and conclusions obtained in our experiments. They contribute nevertheless insights on the root causes of the problem and how to go about alleviating it.

5.1 Summary of Contributions

First we examined the influence of the load-store unit of the PowerPC 7448 on the analysis efficiency and precision by simplifying the hardware model. Our experiments contribute empirical evidence that the deep queues in the load-store unit decrease the efficiency of the WCET analysis by a large factor, while not increasing the system's WCET performance proportionally. Moreover, using a less precise yet much more efficient global-bound analysis (ILP), the computed WCET bounds actually *decrease* (i.e. the WCET analysis predicts *faster* program executions on the simplified platform.)

Building on these observations, we have constructed a compiler optimization which makes programs more analyzable (i.e. improves their analysis efficiency) while running on a commercial hardware architecture. The improvement comes at the cost of a slight increase in the computed WCET bounds when using a precise global-bound analysis variant. Using traditional ILP based global-bound analysis on the other hand yields *lower* WCET bounds for optimized programs (i.e. the compiler optimization improves the precision of the WCET analysis.)

5.2 Conclusions

From the results obtained in this thesis, we conclude the following directions to the designers of future hardware architectures for real-time systems:

- Accommodating too many memory accesses harms the system analyzability while it rarely increases the system performance in the worst case on single-threaded processors.
- It is potentially feasible to incorporate a normalization mechanism in the instruction set architecture, such that the analysis can be normalized at arbitrary program points without the overhead of explicitly adding special instructions.

Future Work

The compiler optimization presented in Chapter 4 is a promising direction because of its applicability to existing hardware architectures and its potential

to achieve significant analysis improvement. The optimization as it is now lacks a good prediction of its effect on the WCET bound. Deriving a precise heuristic to make such prediction would bring us closer to get the best of the two worlds: the analyzability of compositional processors and the performance of modern ones.

Bibliography

[A3m] *AbsInt Advanced Analyzer for PowerPC MPC7448 (Simple Memory Model): User Documentation*. AbsInt Angewandte Informatik GmbH. URL: http://www.absint.com/ait/mpc7448.htm.

[BR07] Claire Burguière and Christine Rochange. "On the Complexity of Modeling Dynamic Branch Predictors when Computing Worst-Case Execution Time". In: *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*. 2007. URL: ftp://ftp.irit.fr/IRIT/TRACES/8158_ercim.pdf.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[Clp] *COIN-OR Linear Programming*. Computational Infrastructure for Operations Research. URL: http://www.coin-or.org/projects/Clp.xml.

[Cul13] Christoph Cullmann. "Cache Persistence Analysis: Theory and Practice". In: *ACM Trans. Embed. Comput. Syst.* 12.1s (2013), 40:1–40:25. ISSN: 1539-9087. DOI: 10.1145/2435227.2435236. URL: http://doi.acm.org/10.1145/2435227.2435236.

[DR11] Fred L. Drake and Guido Rossum. *The Python Language Reference Manual*. Network theory Ltd., 2011. ISBN: 9781906966140. URL: http://www.worldcat.org/isbn/9781906966140.

[EG97] Andreas Ermedahl and Jan Gustafsson. "Deriving Annotations for Tight Calculation of Execution Time". In: *Euro-Par*. 1997, pp. 1298–1307. DOI: 10.1007/BFb0002886.

[EL07] Stephen A. Edwards and Edward A. Lee. "The Case for the Precision Timed (PRET) Machine". In: *DAC*. IEEE, 2007, pp. 264–265.

[Eng02] Jakob Engblom. "Processor Pipelines and Static Worst-Case Execution Time Analysis". PhD thesis. Uppsala University, Sweden, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5355.

[Fer+01] C. Ferdinand et al. "Reliable and Precise WCET Determination for a Real-Life Processor". In: *International Conference on Embedded Software*. Vol. 2211. LNCS. 2001, pp. 469–485. DOI: 10.1007/3-540-45449-7_32.

[FL10] Heiko Falk and Paul Lokuciejewski. "A compiler framework for the reduction of worst-case execution times". In: *Journal on Real-Time Systems* 46.2 (2010), pp. 251–300. DOI: 10.1007/s11241-010-9101-x.

[Fri02] Michael Friendly. *Corrgrams: Exploratory displays for correlation matrices*. 2002.

[FW99] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-Time Systems". In: *Real-Time Sys.* 17.2-3 (1999), pp. 131–181.

[Gcc] *GNU GCC Manual*. Free Software Foundation. 2005. URL: http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/.

[GRG11] Daniel Grund, Jan Reineke, and Gernot Gebhard. "Branch Target Buffers: WCET Analysis Framework and Timing Predictability". In: *Journal of Systems Architecture* 57.6 (2011), pp. 625–637. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.05.013. URL: http://rw4.cs.uni-saarland.de/~grund/papers/jsa10-BTBs.pdf.

[Gus+10] Jan Gustafsson et al. “The Mälardalen WCET Benchmarks – Past, Present and Future”. In: ed. by Björn Lisper. Brussels, Belgium: OCG, July 2010, pp. 137–147. DOI: 10.4230/OASIcs.WCET.2010.136.

[Hea+00] C. Healy et al. “Supporting Timing Analysis by Automatic Bounding of Loop Iterations”. In: *Real-Time Sys.* (2000), pp. 129–156. DOI: 10.1023/A:1008189014032.

[Hec+03] R. Heckmann et al. “The influence of processor architecture on the design and the results of WCET tools”. In: *Proceedings of the IEEE* 91.7 (2003), pp. 1038–1054. ISSN: 0018-9219. DOI: 10.1109/JPROC.2003.814618.

[Hen00] John L. Henning. “SPEC CPU2000: Measuring CPU Performance in the New Millennium”. In: *Computer* 33.7 (2000), pp. 28–35. ISSN: 0018-9162. DOI: 10.1109/2.869367. URL: http://dx.doi.org/10.1109/2.869367.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123704901.

[HPS12] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. “Worst-case execution time analysis-driven object cache design”. In: *Concurrency and Computation: Practice and Experience* 24.8 (2012), pp. 753–771. ISSN: 1532-0634. DOI: 10.1002/cpe.1763. URL: http://dx.doi.org/10.1002/cpe.1763.

[Liu+12] Isaac Liu et al. “A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance”. In: *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*. 2012. DOI: 10.1109/ICCD.2012.6378622. URL: http://chess.eecs.berkeley.edu/pubs/919.html.

[LM95] Yau-Tsun Steven Li and Sharad Malik. “Performance Analysis of Embedded Software Using Implicit Path Enumeration”. In: *Proceedings of the 32nd ACM/IEEE Design Automation Conference*. 1995, pp. 456–461. DOI: 10.1145/217474.217570.

[LS99] T. Lundqvist and P. Stenstrom. “Timing anomalies in dynamically scheduled microprocessors”. In: *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*. 1999, pp. 12–21. DOI: 10.1109/REAL.1999.818824.

[Mpc] *MPC7450 RISC Microprocessor Family Reference Manual*. Freescale Semiconductor.

[MR12] Mohamed Abdel Maksoud and Jan Reineke. “An Empirical Evaluation of the Influence of the Load-Store Unit on WCET Analysis”. In: *12th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 13–24. ISBN: 978-3-939897-41-5. DOI: 10.4230/OASIcs.WCET.2012.13. URL: http://drops.dagstuhl.de/opus/volltexte/2012/3553.

[MR14] Mohamed Abdel Maksoud and Jan Reineke. “A Compiler Optimization to Increase the Efficiency of WCET Analysis”. In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS ’14. Versaille, France: ACM, 2014, 87:87–87:96. ISBN: 978-1-4503-2727-5. DOI: 10.1145/2659787.2659825. URL: http://doi.acm.org/10.1145/2659787.2659825.

[Rei+06] Jan Reineke et al. “A Definition and Classification of Timing Anomalies”. In: *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*. 2006. DOI: 10.1.1.165.9737. URL: http://rw4.cs.uni-saarland.de/~reineke/publications/TimingAnomaliesWCET06.pdf.

[Rei08] Jan Reineke. “Caches in WCET Analysis”. PhD thesis. Universität des Saarlandes, 2008. URL: http://www.rw.cdl.uni-saarland.de/private/reineke/publications/DissertationCachesInWCETAnalysis.pdf.

[RS05] Christine Rochange and Pascal Sainrat. “A Time-predictable Execution Mode for Superscalar Pipelines with Instruction Prescheduling”. In: *Proceedings of the 2nd Conference on Computing Frontiers*. CF ’05. Ischia, Italy: ACM, 2005, pp. 307–314. ISBN: 1-59593-019-1. DOI: 10.1145/1062261.1062312. URL: http://doi.acm.org/10.1145/1062261.1062312.

[RS09] Jan Reineke and Rathijit Sen. “Sound and Efficient WCET Analysis in the Presence of Timing Anomalies”. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11. ISBN: 978-3-939897-14-9. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2009.2289. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2289.

[Sch03] Martin Schoeberl. “JOP: A Java Optimized Processor”. In: *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*. Vol. 2889. Springer, 2003, pp. 346–359. DOI: 10.1007/b94345. URL: http://www.jopdesign.com/doc/jtres03.pdf.

[Sch08] Martin Schoeberl. “A Java processor architecture for embedded real-time systems”. In: *Journal of Systems Architecture* 54.1-2 (2008), pp. 265 –286. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2007.06.001. URL: http://www.sciencedirect.com/science/article/pii/S1383762107000963.

[SM07] Ingmar Stein and Florian Martin. “Analysis of path exclusion at the machine code level”. In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Ed. by Christine Rochange. Vol. 6. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. ISBN: 978-3-939897-05-7. DOI: 10.4230/OASIcs.WCET.2007.1196. URL: http://drops.dagstuhl.de/opus/volltexte/2007/1196.

[The+03] S. Thesing et al. “An abstract interpretation-based timing validation of hard real-time avionics software”. In: *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*. 2003, pp. 625–632. DOI: 10.1109/DSN.2003.1209972.

[The02a] Henrik Theiling. “Control-Flow Graphs For Real-Time Systems Analysis”. PhD thesis. Saarbrücken, Germany: Saarland University, 2002. URL: http://scidok.sulb.uni-saarland.de/volltexte/2004/297/.

[The02b] Henrik Theiling. “ILP-Based Interprocedural Path Analysis”. In: *International Conference on Embedded Software*. Vol. 2491. LNCS. Springer, 2002, pp. 349–363. DOI: 10.1007/3-540-45828-X_26.

[The04] Stephan Thesing. “Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models”. PhD thesis. Saarbrücken, Germany: Saarland University, 2004. URL: http://scidok.sulb.uni-saarland.de/volltexte/2005/466/.

[Tur36] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *J. of Math* 58 (1936), pp. 345–363.

[TW04] Lothar Thiele and Reinhard Wilhelm. “Design for Timing Predictability”. In: *Real-Time Systems* 28 (2004), pp. 157–177. DOI: 10.1023/B:TIME.0000045316.66276.6e.

[Ung+10] Theo Ungerer et al. “Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability”. In: *IEEE Micro* 30.5 (2010), pp. 66–75.

[Wil+09] Reinhard Wilhelm et al. “Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems”. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 28.7 (2009), pp. 966–978. DOI: 10.1109/TCAD.2009.2013287.

[Fre05] Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture*. MPCFPE32B. Specification. 2005. URL: http://www.freescale.com/files/product/doc/MPCFPE32B.pdf.

APPENDIX

A Compiler Optimization Results

This appendix lists the detailed results obtained using the compiler optimization presented in Chapter 4 with various aggressiveness values and using both path analysis variants.

A.1 Using the Prediction-File-Based ILP Path Analysis

WCET bounds and performance metrics for aggressiveness=10% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	567847	0.9999	13404.45	34343.63	2.5621	8186	6747	0.8242
cnt	21897	25157	1.1489	88487.60	27.05	0.0003	21594	102	0.0047
crc	353724	355267	1.0044	24.12	15.20	0.6302	207	185	0.8937
expint	12716	12911	1.0153	2.41	2.94	1.2195	59	58	0.9831
fac	3404	3875	1.1384	0.93	1.07	1.1499	58	58	1.0000
fdct	33637	34222	1.0174	146.50	188.62	1.2876	849	1118	1.3168
fibcall	3387	3387	1.0000	0.48	0.48	1.0042	0	0	1.0000
janne	1793	1751	0.9766	0.86	0.78	0.9043	58	57	0.9828
ludcmp	17750	17787	1.0021	16.99	16.22	0.9548	101	104	1.0297
prime	6801	6802	1.0001	6.25	2.37	0.3795	75	66	0.8800
qurt	18900	18709	0.9899	271.20	147.97	0.5456	822	494	0.6010
ud	14867	14944	1.0052	11.73	9.69	0.8260	83	75	0.9036

WCET bounds and performance metrics for aggressiveness=20% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	568151	1.0004	13404.45	2746.58	0.2049	8186	3113	0.3803
cnt	21897	25438	1.1617	88487.60	24.23	0.0003	21594	92	0.0043
crc	353724	354465	1.0021	24.12	18.07	0.7491	207	175	0.8454
expint	12716	12890	1.0137	2.41	2.93	1.2174	59	58	0.9831
fac	3404	3998	1.1745	0.93	0.70	0.7551	58	57	0.9828
fdct	33637	35513	1.0558	146.50	106.43	0.7265	849	467	0.5501
fibcall	3387	3387	1.0000	0.48	0.48	1.0084	0	0	1.0000
janne	1793	1751	0.9766	0.86	0.77	0.8950	58	57	0.9828
ludcmp	17750	18856	1.0623	16.99	10.04	0.5908	101	83	0.8218
prime	6801	6876	1.0110	6.25	2.35	0.3752	75	66	0.8800
qurt	18900	18513	0.9795	271.20	167.76	0.6186	822	562	0.6837
ud	14867	15366	1.0336	11.73	5.05	0.4305	83	66	0.7952

WCET bounds and performance metrics for aggressiveness=30% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	592601	1.0435	13404.45	64.89	0.0048	8186	643	0.0785
cnt	21897	25610	1.1696	88487.60	24.17	0.0003	21594	91	0.0042
crc	353724	354858	1.0032	24.12	13.75	0.5700	207	176	0.8502
expint	12716	12800	1.0066	2.41	2.98	1.2378	59	58	0.9831
fac	3404	3998	1.1745	0.93	0.70	0.7508	58	57	0.9828
fdct	33637	34589	1.0283	146.50	14.93	0.1019	849	76	0.0895
fibcall	3387	3356	0.9908	0.48	0.41	0.8476	0	0	1.0000
janne	1793	1743	0.9721	0.86	0.87	1.0117	58	58	1.0000
ludcmp	17750	19227	1.0832	16.99	6.50	0.3824	101	75	0.7426
prime	6801	6876	1.0110	6.25	2.34	0.3744	75	66	0.8800
qurt	18900	18656	0.9871	271.20	153.17	0.5648	822	467	0.5681
ud	14867	15612	1.0501	11.73	4.34	0.3699	83	66	0.7952

WCET bounds and performance metrics for aggressiveness=40% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	592295	1.0430	13404.45	69.36	0.0052	8186	701	0.0856
cnt	21897	25927	1.1840	88487.60	24.41	0.0003	21594	95	0.0044
crc	353724	357144	1.0097	24.12	18.54	0.7685	207	179	0.8647
expint	12716	12824	1.0085	2.41	2.67	1.1058	59	66	1.1186
fac	3404	4021	1.1813	0.93	0.46	0.4995	58	0	0.0000
fdct	33637	35158	1.0452	146.50	17.11	0.1168	849	75	0.0883
fibcall	3387	3356	0.9908	0.48	0.39	0.8184	0	0	1.0000
janne	1793	2083	1.1617	0.86	0.76	0.8845	58	57	0.9828
ludcmp	17750	18986	1.0696	16.99	2.88	0.1693	101	67	0.6634
prime	6801	6911	1.0162	6.25	1.92	0.3072	75	58	0.7733
qurt	18900	18496	0.9786	271.20	69.18	0.2551	822	294	0.3577
ud	14867	15835	1.0651	11.73	3.94	0.3357	83	67	0.8072

WCET bounds and performance metrics for aggressiveness=50% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	592673	1.0436	13404.45	74.97	0.0056	8186	771	0.0942
cnt	21897	26262	1.1993	88487.60	25.26	0.0003	21594	97	0.0045
crc	353724	357298	1.0101	24.12	18.20	0.7546	207	177	0.8551
expint	12716	12755	1.0031	2.41	1.22	0.5066	59	69	1.1695
fac	3404	4063	1.1936	0.93	0.46	0.4930	58	0	0.0000
fdct	33637	36084	1.0727	146.50	13.03	0.0890	849	77	0.0907
fibcall	3387	3356	0.9908	0.48	0.39	0.8100	0	0	1.0000
janne	1793	2177	1.2142	0.86	0.86	1.0047	58	58	1.0000
ludcmp	17750	19132	1.0779	16.99	2.75	0.1618	101	67	0.6634
prime	6801	6911	1.0162	6.25	1.84	0.2936	75	58	0.7733
qurt	18900	18775	0.9934	271.20	61.66	0.2274	822	258	0.3139
ud	14867	15868	1.0673	11.73	2.04	0.1744	83	66	0.7952

WCET bounds and performance metrics for aggressiveness=60% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	683014	1.2027	13404.45	64.91	0.0048	8186	717	0.0876
cnt	21897	26285	1.2004	88487.60	24.73	0.0003	21594	101	0.0047
crc	353724	385375	1.0895	24.12	12.77	0.5294	207	175	0.8454
expint	12716	12729	1.0010	2.41	1.23	0.5104	59	69	1.1695
fac	3404	4063	1.1936	0.93	0.46	0.4919	58	0	0.0000
fdct	33637	36594	1.0879	146.50	19.01	0.1298	849	77	0.0907
fibcall	3387	3356	0.9908	0.48	0.39	0.8142	0	0	1.0000
janne	1793	2177	1.2142	0.86	0.86	1.0070	58	58	1.0000
ludcmp	17750	19149	1.0788	16.99	3.14	0.1848	101	66	0.6535
prime	6801	6909	1.0159	6.25	1.82	0.2907	75	58	0.7733
qurt	18900	19724	1.0436	271.20	32.37	0.1194	822	164	0.1995
ud	14867	16127	1.0848	11.73	1.81	0.1542	83	58	0.6988

WCET bounds and performance metrics for aggressiveness=70% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	683120	1.2029	13404.45	68.18	0.0051	8186	693	0.0847
cnt	21897	22627	1.0333	88487.60	21.03	0.0002	21594	93	0.0043
crc	353724	393927	1.1137	24.12	13.32	0.5524	207	179	0.8647
expint	12716	12707	0.9993	2.41	1.15	0.4788	59	58	0.9831
fac	3404	4230	1.2427	0.93	0.46	0.4908	58	0	0.0000
fdct	33637	36965	1.0989	146.50	18.61	0.1271	849	83	0.0978
fibcall	3387	3356	0.9908	0.48	0.39	0.8121	0	0	1.0000
janne	1793	2244	1.2515	0.86	0.57	0.6628	58	0	0.0000
ludcmp	17750	19371	1.0913	16.99	3.24	0.1906	101	66	0.6535
prime	6801	7018	1.0319	6.25	1.91	0.3051	75	58	0.7733
qurt	18900	20351	1.0768	271.20	26.80	0.0988	822	150	0.1825
ud	14867	16560	1.1139	11.73	1.71	0.1458	83	58	0.6988

WCET bounds and performance metrics for aggressiveness=80% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	686466	1.2088	13404.45	80.05	0.0060	8186	821	0.1003
cnt	21897	26703	1.2195	88487.60	19.02	0.0002	21594	83	0.0038
crc	353724	400465	1.1321	24.12	13.12	0.5441	207	177	0.8551
expint	12716	14077	1.1070	2.41	0.94	0.3909	59	58	0.9831
fac	3404	4230	1.2427	0.93	0.46	0.4941	58	0	0.0000
fdct	33637	36882	1.0965	146.50	14.98	0.1023	849	75	0.0883
fibcall	3387	3373	0.9959	0.48	0.40	0.8351	0	0	1.0000
janne	1793	2287	1.2755	0.86	0.71	0.8261	58	57	0.9828
ludcmp	17750	19480	1.0975	16.99	2.52	0.1484	101	66	0.6535
prime	6801	7018	1.0319	6.25	1.91	0.3058	75	58	0.7733
qurt	18900	21036	1.1130	271.20	27.07	0.0998	822	128	0.1557
ud	14867	16888	1.1359	11.73	1.65	0.1409	83	58	0.6988

WCET bounds and performance metrics for aggressiveness=90% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	686466	1.2088	13404.45	79.97	0.0060	8186	821	0.1003
cnt	21897	26703	1.2195	88487.60	18.42	0.0002	21594	90	0.0042
crc	353724	400419	1.1320	24.12	13.04	0.5407	207	176	0.8502
expint	12716	14134	1.1115	2.41	0.94	0.3909	59	58	0.9831
fac	3404	4450	1.3073	0.93	0.45	0.4811	58	0	0.0000
fdct	33637	37712	1.1211	146.50	18.04	0.1232	849	75	0.0883
fibcall	3387	3373	0.9959	0.48	0.40	0.8309	0	0	1.0000
janne	1793	2287	1.2755	0.86	0.71	0.8273	58	57	0.9828
ludcmp	17750	19638	1.1064	16.99	2.04	0.1202	101	59	0.5842
prime	6801	7122	1.0472	6.25	1.55	0.2482	75	58	0.7733
qurt	18900	20968	1.1094	271.20	19.43	0.0716	822	105	0.1277
ud	14867	17042	1.1463	11.73	1.72	0.1468	83	58	0.6988

WCET bounds and performance metrics for aggressiveness=100% (PF-ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	567899	686466	1.2088	13404.45	79.82	0.0060	8186	821	0.1003
cnt	21897	28961	1.3226	88487.60	23.31	0.0003	21594	84	0.0039
crc	353724	405135	1.1453	24.12	13.48	0.5586	207	177	0.8551
expint	12716	14143	1.1122	2.41	0.96	0.4004	59	58	0.9831
fac	3404	4433	1.3023	0.93	0.44	0.4746	58	0	0.0000
fdct	33637	38106	1.1329	146.50	15.76	0.1075	849	76	0.0895
fibcall	3387	3373	0.9959	0.48	0.40	0.8246	0	0	1.0000
janne	1793	2305	1.2856	0.86	0.70	0.8191	58	58	1.0000
ludcmp	17750	20222	1.1393	16.99	2.04	0.1200	101	58	0.5743
prime	6801	7702	1.1325	6.25	1.51	0.2412	75	58	0.7733
qurt	18900	21081	1.1154	271.20	16.63	0.0613	822	102	0.1241
ud	14867	17055	1.1472	11.73	1.71	0.1456	83	58	0.6988

A.2 Using the Traditional ILP Path Analysis

WCET bounds and performance metrics for aggressiveness=10% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	682543	0.9980	832.81	395.53	0.4749	6873	5182	0.7540
cnt	40422	27909	0.6904	9596.36	18.80	0.0020	3340	90	0.0269
crc	361219	357123	0.9887	15.65	11.69	0.7469	211	188	0.8910
expint	13543	13772	1.0169	1.99	2.21	1.1094	58	58	1.0000
fac	4357	4260	0.9777	0.72	1.06	1.4792	58	59	1.0172
fdct	36459	37145	1.0188	21.86	18.83	0.8613	98	98	1.0000
fibcall	3793	3793	1.0000	0.42	0.43	1.0240	0	0	1.0000
janne	2313	2225	0.9620	0.60	0.62	1.0331	57	58	1.0175
ludcmp	21945	21242	0.9680	6.16	5.34	0.8660	98	91	0.9286
prime	8354	7520	0.9002	3.58	1.49	0.4164	74	58	0.7838
qurt	27935	25004	0.8951	45.72	21.57	0.4718	266	171	0.6429
ud	18660	18343	0.9830	4.96	4.20	0.8459	82	74	0.9024

WCET bounds and performance metrics for aggressiveness=20% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	763564	1.1164	832.81	163.94	0.1968	6873	2728	0.3969
cnt	40422	27671	0.6846	9596.36	17.76	0.0019	3340	90	0.0269
crc	361219	356600	0.9872	15.65	11.13	0.7111	211	179	0.8483
expint	13543	13630	1.0064	1.99	2.17	1.0888	58	58	1.0000
fac	4357	4236	0.9722	0.72	0.90	1.2514	58	58	1.0000
fdct	36459	37717	1.0345	21.86	11.16	0.5107	98	82	0.8367
fibcall	3793	3793	1.0000	0.42	0.42	1.0024	0	0	1.0000
janne	2313	2225	0.9620	0.60	0.62	1.0331	57	58	1.0175
ludcmp	21945	22074	1.0059	6.16	4.19	0.6794	98	82	0.8367
prime	8354	7563	0.9053	3.58	1.52	0.4259	74	58	0.7838
qurt	27935	24216	0.8669	45.72	22.80	0.4987	266	170	0.6391
ud	18660	18111	0.9706	4.96	3.36	0.6767	82	66	0.8049

WCET bounds and performance metrics for aggressiveness=30% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	787518	1.1514	832.81	37.96	0.0456	6873	646	0.0940
cnt	40422	28358	0.7015	9596.36	17.44	0.0018	3340	90	0.0269
crc	361219	356105	0.9858	15.65	10.95	0.6995	211	180	0.8531
expint	13543	13518	0.9982	1.99	2.23	1.1209	58	58	1.0000
fac	4357	4236	0.9722	0.72	0.90	1.2500	58	58	1.0000
fdct	36459	36975	1.0142	21.86	10.75	0.4916	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.35	0.8345	0	0	1.0000
janne	2313	2254	0.9745	0.60	0.63	1.0381	57	58	1.0175
ludcmp	21945	21976	1.0014	6.16	2.83	0.4595	98	74	0.7551
prime	8354	7563	0.9053	3.58	1.53	0.4287	74	66	0.8919
qurt	27935	24393	0.8732	45.72	21.76	0.4760	266	154	0.5789
ud	18660	18005	0.9649	4.96	3.09	0.6220	82	67	0.8171

WCET bounds and performance metrics for aggressiveness=40% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	767803	1.1226	832.81	40.87	0.0491	6873	702	0.1021
cnt	40422	28654	0.7089	9596.36	17.70	0.0018	3340	90	0.0269
crc	361219	358287	0.9919	15.65	10.34	0.6607	211	178	0.8436
expint	13543	13343	0.9852	1.99	1.51	0.7597	58	58	1.0000
fac	4357	4277	0.9816	0.72	0.42	0.5847	58	0	0.0000
fdct	36459	37420	1.0264	21.86	11.50	0.5260	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.35	0.8489	0	0	1.0000
janne	2313	2339	1.0112	0.60	0.57	0.9421	57	57	1.0000
ludcmp	21945	20798	0.9477	6.16	1.84	0.2987	98	66	0.6735
prime	8354	7435	0.8900	3.58	1.29	0.3619	74	58	0.7838
qurt	27935	23604	0.8450	45.72	14.02	0.3066	266	130	0.4887
ud	18660	18162	0.9733	4.96	2.92	0.5887	82	58	0.7073

WCET bounds and performance metrics for aggressiveness=50% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	777831	1.1373	832.81	40.49	0.0486	6873	774	0.1126
cnt	40422	29130	0.7206	9596.36	18.13	0.0019	3340	90	0.0269
crc	361219	358246	0.9918	15.65	9.85	0.6295	211	180	0.8531
expint	13543	13242	0.9778	1.99	1.02	0.5133	58	58	1.0000
fac	4357	4319	0.9913	0.72	0.41	0.5667	58	0	0.0000
fdct	36459	38839	1.0653	21.86	9.33	0.4269	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.35	0.8393	0	0	1.0000
janne	2313	2586	1.1180	0.60	0.64	1.0613	57	57	1.0000
ludcmp	21945	20815	0.9485	6.16	1.70	0.2760	98	67	0.6837
prime	8354	7445	0.8912	3.58	1.34	0.3737	74	58	0.7838
qurt	27935	23382	0.8370	45.72	14.33	0.3134	266	130	0.4887
ud	18660	17251	0.9245	4.96	1.59	0.3212	82	66	0.8049

WCET bounds and performance metrics for aggressiveness=60% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	887372	1.2974	832.81	34.14	0.0410	6873	718	0.1045
cnt	40422	29097	0.7198	9596.36	17.79	0.0019	3340	90	0.0269
crc	361219	386128	1.0690	15.65	9.71	0.6207	211	179	0.8483
expint	13543	13211	0.9755	1.99	1.03	0.5168	58	58	1.0000
fac	4357	4319	0.9913	0.72	0.42	0.5861	58	0	0.0000
fdct	36459	38765	1.0632	21.86	11.78	0.5388	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.37	0.8801	0	0	1.0000
janne	2313	2586	1.1180	0.60	0.65	1.0695	57	57	1.0000
ludcmp	21945	20908	0.9527	6.16	1.79	0.2904	98	67	0.6837
prime	8354	7420	0.8882	3.58	1.30	0.3633	74	58	0.7838
qurt	27935	23101	0.8270	45.72	8.54	0.1868	266	90	0.3383
ud	18660	17170	0.9202	4.96	1.39	0.2796	82	58	0.7073

WCET bounds and performance metrics for aggressiveness=70% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	892309	1.3047	832.81	33.86	0.0407	6873	693	0.1008
cnt	40422	24870	0.6153	9596.36	14.35	0.0015	3340	90	0.0269
crc	361219	394918	1.0933	15.65	10.17	0.6499	211	179	0.8483
expint	13543	13081	0.9659	1.99	0.94	0.4696	58	58	1.0000
fac	4357	4471	1.0262	0.72	0.41	0.5667	58	0	0.0000
fdct	36459	39264	1.0769	21.86	11.84	0.5416	98	82	0.8367
fibcall	3793	3648	0.9618	0.42	0.52	1.2350	0	0	1.0000
janne	2313	2616	1.1310	0.60	0.49	0.8046	57	0	0.0000
ludcmp	21945	21105	0.9617	6.16	1.85	0.3003	98	67	0.6837
prime	8354	7514	0.8994	3.58	1.29	0.3594	74	58	0.7838
qurt	27935	23716	0.8490	45.72	9.78	0.2139	266	90	0.3383
ud	18660	17466	0.9360	4.96	1.36	0.2733	82	58	0.7073

WCET bounds and performance metrics for aggressiveness=80% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	1104873	1.6154	832.81	45.20	0.0543	6873	822	0.1196
cnt	40422	29170	0.7216	9596.36	13.56	0.0014	3340	93	0.0278
crc	361219	401339	1.1111	15.65	10.19	0.6509	211	180	0.8531
expint	13543	14379	1.0617	1.99	0.81	0.4064	58	58	1.0000
fac	4357	4471	1.0262	0.72	0.42	0.5875	58	0	0.0000
fdct	36459	39452	1.0821	21.86	10.43	0.4770	98	74	0.7551
fibcall	3793	3660	0.9649	0.42	0.40	0.9688	0	0	1.0000
janne	2313	2540	1.0981	0.60	0.56	0.9305	57	57	1.0000
ludcmp	21945	20799	0.9478	6.16	1.52	0.2468	98	58	0.5918
prime	8354	7514	0.8994	3.58	1.29	0.3619	74	58	0.7838
qurt	27935	23947	0.8572	45.72	7.33	0.1603	266	82	0.3083
ud	18660	17868	0.9576	4.96	1.32	0.2661	82	58	0.7073

WCET bounds and performance metrics for aggressiveness=90% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	1104873	1.6154	832.81	44.31	0.0532	6873	821	0.1195
cnt	40422	29170	0.7216	9596.36	13.77	0.0014	3340	82	0.0246
crc	361219	401348	1.1111	15.65	10.45	0.6676	211	178	0.8436
expint	13543	14407	1.0638	1.99	0.82	0.4104	58	58	1.0000
fac	4357	4695	1.0776	0.72	0.50	0.6931	58	0	0.0000
fdct	36459	40220	1.1032	21.86	11.34	0.5188	98	74	0.7551
fibcall	3793	3660	0.9649	0.42	0.37	0.8873	0	0	1.0000
janne	2313	2540	1.0981	0.60	0.56	0.9321	57	57	1.0000
ludcmp	21945	21045	0.9590	6.16	1.42	0.2299	98	59	0.6020
prime	8354	7690	0.9205	3.58	1.12	0.3133	74	58	0.7838
qurt	27935	23812	0.8524	45.72	5.60	0.1225	266	82	0.3083
ud	18660	17929	0.9608	4.96	1.39	0.2804	82	58	0.7073

WCET bounds and performance metrics for aggressiveness=100% (ILP).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Mod.	Mod.	Ratio	¬Mod.	Mod.	Speedup	¬Mod.	Mod.	Ratio
bsort100	683942	1104873	1.6154	832.81	44.65	0.0536	6873	822	0.1196
cnt	40422	31469	0.7785	9596.36	13.66	0.0014	3340	90	0.0269
crc	361219	405855	1.1236	15.65	10.02	0.6403	211	177	0.8389
expint	13543	14415	1.0644	1.99	0.81	0.4089	58	58	1.0000
fac	4357	4678	1.0737	0.72	0.56	0.7764	58	58	1.0000
fdct	36459	40758	1.1179	21.86	11.16	0.5108	98	74	0.7551
fibcall	3793	3660	0.9649	0.42	0.37	0.8825	0	0	1.0000
janne	2313	2557	1.1055	0.60	0.56	0.9288	57	57	1.0000
ludcmp	21945	21527	0.9810	6.16	1.43	0.2324	98	59	0.6020
prime	8354	8204	0.9820	3.58	1.19	0.3331	74	58	0.7838
qurt	27935	24005	0.8593	45.72	5.64	0.1233	266	82	0.3083
ud	18660	17956	0.9623	4.96	1.35	0.2721	82	58	0.7073