Thesis for the degree of Doctor of Philosophy

A WCET Analysis Method for Pipelined Microprocessors with Cache Memories

Thomas Lundqvist

Department of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2002

A WCET Analysis Method for Pipelined Microprocessors with Cache Memories
Thomas Lundqvist
ISBN 91-7291-182-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 1864
ISSN 0346-718X

School of Computer Science and Engineering
Chalmers University of Technology
Technical report 2D

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

Author email address: tlundqvist@acm.org

Printed by Chalmers Reproservice
Göteborg, Sweden 2002

A WCET Analysis Method for Pipelined Microprocessors with Cache Memories

Thomas Lundqvist
Department of Computer Engineering, Chalmers University of Technology

Abstract

When constructing real-time systems, safe and tight estimations of the worst case execution time (WCET) of programs are needed. To obtain tight estimations, a common approach is to do path and timing analyses. Path analysis is responsible for eliminating infeasible paths in the program and timing analysis is responsible for accurately modeling the timing behavior of programs. The focus of this thesis is on analysis of programs running on high-performance microprocessors employing pipelining and caching.

This thesis presents a new method, referred to as *cycle-level symbolic execution*, that tightly integrates path and timing analysis. An implementation of the method has been used to estimate the WCET for a suite of programs running on a high-performance processor. The results show that by using an integrated analysis, the overestimation is significantly reduced compared to other methods. The method automatically eliminates infeasible paths and derives path information such as loop bounds, and performs accurate timing analysis for a multiple-issue processor with an instruction and data cache. The thesis also identifies timing anomalies in dynamically scheduled processors. These anomalies can lead to unbounded timing effects when estimating the WCET, which makes it unsafe to use previously presented timing analysis methods. To handle these unbounded timing effects, two methods are proposed. The first method is based on program modifications and the second method relies on using pessimistic timing models. Both methods make it possible to safely use all previously published timing analysis methods even for architectures where timing anomalies can occur. Finally, the use of data caching is examined. For data caching to be fruitful in real-time systems, data accesses must be predictable when estimating the WCET. Based on a notion of predictable and unpredictable data structures, it is shown how to classify program data structures according to their influence on data cache analysis. For both categories, several examples of frequently used types of data structures are provided. Furthermore, it is shown how to make an efficient data cache analysis even when data structures have an unknown placement in memory. This is important, for example, when analyzing single subroutines of a program.

Keywords: Real-time systems, worst-case execution time, timing analysis, path analysis, infeasible paths, pipeline, instruction cache, data cache, timing anomaly, dynamically scheduled processor.

Contents

Preface

This thesis builds upon research done by me since 1996 when I started my Ph.D studies. In 1996, my advisor Per Stenström had an idea about using instruction-level simulation for estimating WCET. Not knowing better, I though this was a excellent idea and started to work on the subject. The first year was spent trying to explain to Per why the approach would not work. Luckily, Per did not believe me and I finally stumbled upon the method presented in this thesis.

The problem of estimating the WCET of a program is a really good problem. It is quite complicated and involves many different areas: computer architucture, programming languages, operating systems, program analysis, and software engineering. Partly due to the complexity, I believe, this area of research has been growing for the last 10 years. To do research in such an area has been an exciting experience.

To finish your Ph.D. studies, you need a good and friendly working environment to support you. I can think of no better place than the Computer Engineering department here at Chalmers. It is filled with highly skilled people that also are friendly—a winning combination.

My thanks go first to my advisor Professor Per Stenström, for the dedicated support and excellent guidance given me during all these years. Even if the research behind this thesis has been carried out by me, his criticism and demand for consistency have pushed this research further than what would have otherwise been possible. Also, as a coauthor of many of my publications, he has used his magic hands to make the presentation of the results comprehensible.

Further thanks go to the people that always are or have been willing to listen to my problems and solutions: Jim Nilsson, who knows everything and can see through all your fake arguments; Peter Rundberg, the hardware guru who also knows everything; Björn Andersson,

who loves wild discussions about WCET, life, the universe, and everything; Fredrik Warg, the discreet northerner who always accept my need for discussions; Jochen Hollman, who adds that German rigor; Cecilia Ekelin, the constraint solving expert; Håkan Forsberg, who always can update you with the latest high-tech news; Jonas Jalminger, who always give you answers with a smile; Robert Feldt, who can answer all your more software oriented questions; Martin Hiller, who also can answer your software questions but with an added touch of fault tolerance; Jan Jonsson, the scheduling expert who always have that hard-to-get article in his library; Jonas Vasell, who personifies academic reasoning and rigor but always believes that statistical methods for WCET estimation is the future; Magnus Ekman, the smart newcomer who also can see through your fake arguments; and finally some emigrants: Magnus Karlsson, who helped me a lot in the beginning; and Martin Kämpe, who also give you answers with a smile.

There are also many others that have contributed to a supporting and friendly environment: Jonas Lext, Ulf Assarsson, Tomas Möller, Fredrik Dahlgren, Elisabeth Uhlemann, Lars Rasmussen, Ulf Hansson, and Fredrik Lundholm. I am also grateful to the supporting staff that keeps the offices, paper work, and computers running.

Finally, thank you Ulrika, Emil, and Andrea for being there. I will soon be home to hug you.

Chapter 1

Introduction

During the last decades, general-purpose computers have been increasing their performance by several magnitudes. The high performance we see today has been accomplished by using a high processor clock frequency and new architectural mechanisms in the processor. This has also made processors more and more complex. For example, cache memories are used to compensate for the relatively slow speed of memory and instructions are executed in parallel by using pipelined execution and multiple execution units.

In some computer systems, a high performance can be useless if we cannot make guarantees of the performance when designing the system. One example of such systems is real-time systems, where the computer must interact with its environment in a timely manner. For example, a computer controlling the ignition in a combustion engine must finish its calculations before its deadline, i.e., before it is time for the next ignition. A high performance can help a program to finish in time. On the other hand, a too high performance can be a waste of resources. If there is no added benefit of going much faster than the deadline, then a slower, less expensive, and more energy efficient processor can be used.

A program deadline is often classified as being hard or soft depending on what happens if the deadline is missed. If hard deadlines are missed the system will fail, whereas missed soft deadlines only degrades the service provided by the system. When designing a program that has a hard deadline constraint, we must be able to certify that the program finishes within the given deadline before we run it in a production environment.

This means that we must assure that the execution time of the program always is shorter than the specified time. In practice, this is often done by using extensive testing, i.e., by running the program many times with different input data and measuring the execution time. The problem with testing is that it can take a long time to reach the level of confidence that we want. There are often too many combinations of input data to try, and testing can only try relatively few of them. If missing a deadline can lead to very costly consequences, testing may not be an adequate method.

A promising way to improve the situation is to complement the testing with an analysis tool that estimates the worst-case execution time (WCET) of a program. Such a tool can cover all combinations of input data but with lower precision than testing, i.e., the actual WCET can be overestimated but is still safe to use to verify that a program finishes before its deadline. The principles behind one approach to construct such an analysis tool is the topic of this thesis.

1.1 Worst-case execution time

The execution time of a program that runs uninterrupted on a processor depends both on program characteristics and the hardware that executes the program. For a simple processor where single instructions have a fixed execution time, the program execution time is solely determined by the input data given to the program. The input data can influence the path taken through the program and the number of loop iterations done in loops and this causes the execution time to vary.

For a more complex computer system, the execution time also depends on the previous programs that have executed on the processor, the *execution history* of the system. Also, some single instructions can have an execution time that depends on the input data. A good example of the added complexity is to consider a data cache that holds recently fetched data from the main memory. A single load instruction that loads data from memory executes fast if the data already exists in the cache (a cache hit) and slow if it must fetch data from memory (a cache miss). The outcome can depend both on the previous content of the data cache (the execution history) and the address used when accessing memory and this address can depend on the input data given to the program.

So, in order to find the worst-case execution time, we must consider

both the program characteristics and the hardware platform that runs the program:

Definition 1.1 *The worst-case execution time (WCET) of a program that runs uninterrupted on a processor is defined as being the maximum possible execution time considering all combinations of input data and all possible execution histories of the system before the program is executed.*

The WCET of a program can be infinite. In that case, it will surely miss its deadline. We will assume that a program, or part of a program that we want to study, has a finite WCET, at least for the input data range the program is supposed to handle.

1.2 Research goal

The overall goal of the work presented in this thesis is to develop analysis methods that can derive estimates of the WCET for programs running on computer systems where the execution history can influence the timing of instructions.

The estimates derived must be *safe* and also preferably *tight*. A safe estimate means that the actual WCET is not underestimated so that the estimate can be used to verify that a program finishes before its deadline. A tight estimate means that the overestimation should be as small as possible in order to make effective use of the processor.

The target system that we would like to be able to analyze consists of a processor that executes multiple instructions simultaneously in a pipelined manner and uses instruction and data caching. An example of such a system can be seen in Figure 1.1. In each cycle, one or several new instruction are fetched from the instruction cache (or the main memory in case of a cache miss) and inserted into the pipeline. First, the instructions are decoded and then they wait to get dispatched to one of several execution units. The load/store unit handles memory access instructions that transfers data between the processor registers and the data cache (or main memory in case of a cache miss). The integer and the floating point units handle instructions that perform arithmetic operations between the registers.

To reach high performance, several more complex techniques are used. Typically, multiple instructions can be fetched and executed in each cycle.

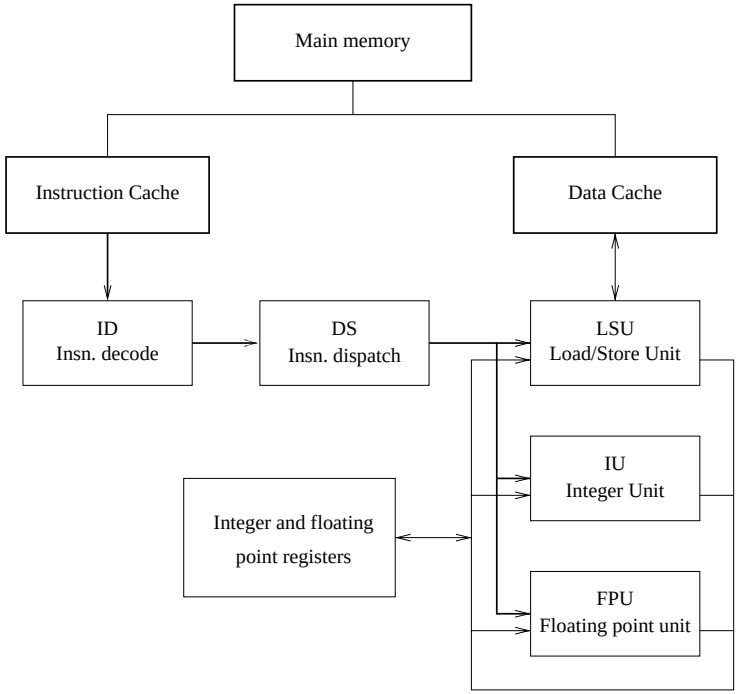


Figure 1.1: Example of pipelined architecture.

Then, because the instruction execution time can vary depending on type and input data, these instructions are allowed to execute out-of-order, i.e., not in program order. Also, the scheduling of instructions can be static or dynamic based on whether the execution order is determined by the program code or determined at run-time. Ideally, the goal is to be able to analyze the impact of these mechanisms on the worst-case execution time.

Often, a program is interrupted (preempted) by other programs sharing the same processor and by interrupt handlers servicing hardware devices. In this case, the WCET estimates of all programs will serve as input to a schedulability analysis that is used to verify that a set of programs, running on the same processor, finishes before their deadlines. The definition of WCET relies on uninterrupted program execution. Therefore, the schedulability analysis must include an analysis of the effects of preemption on the worst-case execution time. Further discussion about scheduling issues can be found in Section 6.6.

1.3 Problem definition

An exact approach to find the WCET, is to analyze the program once for each combination of input data and find the input values that give the maximum execution time. However, an exhaustive search through all input data is often not practical due to the large number of combinations to test. Instead, a common approach is to find the longest path in the program, i.e., the path having the longest worst-case execution time. This approach can cause overestimation of the WCET due to two important reasons. First, the analysis may include program paths that can never be executed regardless of the input data, usually referred to as *infeasible*[1] program paths. Second, the timing model of the hardware platform may introduce overestimations of the WCET because of simplifying timing assumptions. To reduce the overestimation we need to perform good path analysis as well as good timing analysis.

[1]Infeasible paths are sometimes called non-executable or dead program paths

1.3.1 Path analysis

The path analysis is responsible for identifying infeasible paths. Ideally, the set of program paths considered when searching for the longest path should be the same as when an exact exhaustive testing approach is used. However, in the general case, it is not possible to derive this set of feasible paths solely from the program code and still avoid an exhaustive search through input data. Therefore, a common solution is to separately add information about infeasible paths to the WCET analysis. The most important infeasible paths to eliminate are the ones leading to infinite number of loop iterations or recursive function calls. The elimination of other infeasible paths serves to tighten the analysis.

Another complication is that we might need an analysis for only a subset of the input data. A typical reason for this is that the program is only defined for a subset of the input data and might not even terminate for input data outside of this subset. Other reasons include, for example, that we want several analyses for specific subsets of the input data representing different modes of operation for the program. From a path analysis perspective, restricting the analysis to a subset of the input data corresponds to a possible increase of the number of infeasible paths to eliminate.

Information about infeasible paths can be given by the programmer by means of manual path annotations [PK89]. However, this requires a considerable programming effort and is error-prone. A more attractive way is to use methods that do *automatic path analysis* [Alt96, CBW94, EG97, FHL+01, HSR+00, HW99, GL02, NP93, EY97]. These can automatically derive upper bounds on many loops and identify many infeasible paths. Thus, the need for manual annotations can be reduced or completely removed. Furthermore, automatic path analysis methods can make it more easy to express which subset of the input data we want to analyze. A general problem that all methods face is how to combine an automatic path analysis with accurate timing analysis.

1.3.2 Timing analysis

The timing analysis must calculate how many cycles each instruction requires to execute. Since the instruction execution time for an instruction can depend on the other instructions that execute simultaneously in the

pipeline and the execution history, the result gets more accurate when many instructions are analyzed together. Ideally, all instructions along the worst-case path in the program should be analyzed together to obtain the most accurate result. However, this is not generally feasible since we do not know if a path is the worst-case path before having analyzed all paths and the number of program paths to examine can be too many.

For simple architectures, where the execution time of each instruction is independent of the execution history, the methods presented by, for example, Puschner and Koza [PK89] or Shaw [Sha89] can be used. These methods calculate the sum of all execution times for each instruction or statement. However, for more complex architectures, simple arithmetic is not sufficient. When determining the execution time of a single instruction we must take into account how the timing is influenced by all previously executed instruction. To solve this problem, we are forced to use some kind of heuristic approach. A commonly used approach is to split the timing analysis into one analysis for each architectural mechanism.

Previously presented timing analysis approaches [CP01, FHL+01, HWH95, WMH+99, Mue00, LBJ+94, KMH96, LMW95, LMW96, OS97, PS97, SA00, WE00] have targeted instruction and data cache analysis, branch prediction analysis, and pipeline analysis. These methods can be used to successfully analyze a wide range of architectures. However, many problems remain to be solved. It is still unclear if it is possible to make an accurate analysis for a more complex pipeline, using for example, dynamic scheduling of instructions. Furthermore, instruction caching and branch prediction can be predictably analyzed, but data caching poses a more challenging problem. The reason for this is that instruction caching and branch prediction depends on the control flow of the program which is often predictable, while data caching depends on data accesses that can reference data in an order determined dynamically during execution and this order can also depend on input data.

1.4 Contributions

The main contribution in this thesis concerns a new approach to WCET estimation. Based on instruction-level simulation, each path in a program is simulated using a cycle-accurate timing model. The approach integrates

automatic path analysis with timing analysis in a way that can result in very tight WCET estimates. Further contributions focus on the problems of dynamically scheduled pipelines and efficient data cache analysis.

Contributions concerning the new approach:

1. I present a new approach to static estimation of the WCET, referred to as *cycle-level symbolic execution* [LS98, LS99a], that integrates automatic path analysis with timing analysis. I demonstrate how to handle an example system with a dual-issue pipeline with in-order instruction dispatch and instruction and data caches. The memory system beyond the cache memories has not been considered.

2. I have constructed a prototype tool that implements the new approach. The tool works on the object-code level but assumes that the source code is written in C or a similar language.

3. An experimental evaluation has been done by using the prototype tool to estimate the WCET for several benchmark programs. The results show that the method can derive tight estimates of the WCET.

Contributions concerning pipeline analysis:

3. I identify the presence of timing anomalies in dynamically scheduled, pipelined processors where instructions can execute out-of-order [LS99c]. These anomalies can lead to unbounded timing effects that make it unsafe to use my basic approach for dynamically scheduled processors. This unsafe behavior is also a problem for other previously presented timing analysis methods [CP01, FHL+01, HWH95, WMH+99, Mue00, LBJ+94, KMH96, LMW95, LMW96, OS97, PS97, SA00, WE00].

4. I present and evaluate new approaches to avoid these anomalies.

Contributions concerning data cache analysis:

5. Based on my notion of predictable and unpredictable data structures, I show how to classify program data structures according to their influence on data cache analysis [LS99b].

6. I provide examples of frequently used types of data structures that fall into these two categories and also propose a way of making their cache behavior predictable.

7. I show how to make efficient data cache analysis even when data structures have an unknown placement in memory [Lun02]. This is important, for example, when analyzing single subroutines in a program.

1.5 Organization of the thesis

This thesis is organized as follows:

Chapters 2-5 In these chapters, I describe and evaluate my new approach to WCET estimation, cycle-level symbolic execution. Chapter 2 focuses on the path analysis capabilities, explaining how infeasible paths are eliminated and how the merging of paths works. In Chapter 3, a timing model is added. The central problem here is how to perform the merge operation while still deliver a safe estimate. The merge operation is defined for a simple pipeline processor with instruction and data caches. I discuss the implementation of the prototype tool in Chapter 4 before presenting the results from the experimental evaluation in Chapter 5.

Chapter 6 Here, I identify where timing anomalies show up in dynamically scheduled processors. I give examples of several situations where traditional timing analysis methods (mine included) will give an unsafe result. Finally, I suggest and evaluate methods that make it possible to derive safe estimates.

Chapters 7-8 These chapters concern data cache analysis. In Chapter 7, I introduce the notion of when a data structure is considered predictable and show a method to classify data structures as predictable or unpredictable. In Chapter 8 I extend this by also showing how to handle data structures with an unknown placement in memory.

Chapter 9 In this chapter I conclude the thesis by discussing some future and related work.

Chapter 2

Cycle-Level Symbolic Execution

Cycle-level architectural simulation techniques is commonly used for studying the interaction between running programs and architectural mechanisms [ALE02, HPRA02, MDG+98]. It is also used more and more to debug programs, even for real-time systems [AM00]. We propose to also use these techniques to estimate the WCET of a program. The advantages of using architectural simulation techniques are twofold. First, it is possible to make arbitrarily accurate estimations of the execution time of a program for a given set of input data. Second, and presumably more importantly, when a given path through the program is simulated, precise information about this path is automatically extracted.

A cycle-level architectural simulator can be seen as an instruction-level simulator connected to a clock-cycle accurate architectural timing model. In this chapter, we will focus on the path analysis capabilities and assume an instruction-level simulator connected to a timing model that uses fixed instruction execution-times. In the next chapter, which deals with timing analysis, we will show how a more complex timing model can be integrated in this framework.

In this chapter, we first present how simulation can be extended to estimate WCET and the principles behind the automated path analysis. This is done in Sections 2.1–2.3. Then, in Section 2.4 we describe how the merging of paths is done and how this is controlled by the WCET method. In the end of this chapter, in Section 2.6, we relate our approach

to path analysis with previous work done in this area.

2.1 The approach

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

Each data type is extended with an element denoted *unknown*. In general, the semantics of all arithmetics and logical operations must be redefined to correctly calculate the result if any of the source operands have an *unknown* value. Examples of the extended semantics for some common instruction types can be seen in Table 2.1. Consider for example an add instruction, ADD T, A, B, that operates on the set of 32-bit unsigned integers, $Z = \{0 \dots 2^{32} - 1\}$ and is defined as $T \leftarrow A + B$. In the extended semantics, it is instead defined on the set $\overline{Z} = Z \cup \{unknown\}$ with the semantics seen in Table 2.1.

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

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

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

Table 2.1: Extended semantics of instructions.


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

Figure 2.1: Example program that sums some of the values in the unknown matrix b according to the known boolean matrix m.

detail in the next section.

2.2 Path analysis

To understand how the cycle-level symbolic execution technique can automate path analysis, consider the program in Figure 2.1 which calculates the sum of some values in the upper-right triangle of matrix b. For simplicity, we reason about the program in a high-level language, even if the symbolic execution of course is done at the instruction level.

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

is also determined during the simulation.

An important advantage of the symbolic execution approach is that all conditions that depend on data values that are known statically will be computed during the simulation. For example, input data independent bounds on the number of loop iterations that are expressed with arbitrarily complex functions are computed automatically. Interestingly, our method also applies to recursive functions provided that the recursion depth is bounded. Moreover, many infeasible paths will be excluded from the analysis and will therefore not affect the WCET estimation.

2.3 Unknown store instructions

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

We identify the unpredictable data structures by letting the simulator output a list of all unknown stores it encounters. With the help of a source-code debugger, it is possible to manually connect each store instruction to a data structure which is marked as unpredictable. Eventually, the linking phase is redone to properly map the marked data structures to the 'unknown' area of memory. After this step, a correct estimation may be done. The approach mentioned above can be used for statically allocated data structures only. For unpredictable dynamically allocated data structures we do not yet have any solution. A further discussion about predictable and unpredictable data accesses can be found in Chapter 7.

A key problem with the simulation approach in this section is the explosion in number of paths to be simulated. If a loop iteration has n feasible paths and the number of loop iterations is k, the number of paths

to simulate is n^k. Fortunately, good heuristics exist to drastically reduce the number of paths we need to simulate. We have used a path merging strategy, which forms the basis of our WCET method to be presented in the next section.

2.4 The WCET Method

To reduce the number of paths that have to be explored during path analysis, we apply a path-merge strategy. This reduces the number of paths to be explored from n^k down to n for a loop containing n feasible paths in each iteration and doing k iterations. In each loop iteration, all n paths are explored but in the beginning of the next iteration all these n paths are merged into one. Thus, the number of simulated paths is less than or equal to n. We first describe how the merging operation is performed in Section 2.4.1. This operation is used in the WCET algorithm which we present in Section 2.4.2. Finally, in Section 2.4.3 we discuss time complexity and termination of the algorithm.

2.4.1 Merging algorithm

In order to understand how the merging of two paths is done, consider again the example program in Figure 2.1. In the second iteration of the outer loop, when the simulator encounters the unknown conditional branch at line 4, two paths can be merged. When merging, the long path, i.e., the one with the highest WCET, is kept and the short path is discarded. However, to make a valid estimation of the worst-case execution path throughout the program execution, the impact of the short path on the total WCET must be taken into account. For example, variable `sum` can be assigned different values in the two paths. Therefore, all variables whose values differ must be assigned the value *unknown* in the resulting path of the merge operation.

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

$\{PCT$ = program counter, CC = condition code,
R = register, and MEM = memory$\}$
Require: $PCT_A = PCT_B$
$PCT_C \leftarrow PCT_A$

$CC_C \leftarrow CC_A \cup CC_B$
for all registers $R[i]$ **do**
$R_C[i] \leftarrow R_A[i] \cup R_B[i]$
end for
for all memory positions a **do**
$MEM_C[a] \leftarrow MEM_A[a] \cup MEM_B[a]$
end for

$wcet_C \leftarrow \max(wcet_A, wcet_B)$

Figure 2.2: Algorithm merging two paths p_A and p_B creating p_C.

whose values differ. We denote this operation as the *union* operation of path system-states.

The merging algorithm is described in Figure 2.2. It creates a new path p_C from two paths p_A and p_B. The program counter, which must be the same in the two paths merged, is copied to the new path. The new WCET is the maximum of the WCET of the two original paths. Finally, the union of the system states of the merged paths is calculated. The state of a path consists of: $state = \langle PCT, CC, R, MEM \rangle$, where PCT is the program counter, CC is the condition code register, R is the set of processor registers, and MEM is the memory content. The union operation between values is defined as:

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

2.4.2 WCET algorithm

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

The algorithm (see Figure 1) starts the simulation from the beginning of the program. Whenever an unknown conditional branch is found the simulation is stopped and the algorithm selects as the next path to simulate the path that has made the least progress, *a minimum progress path*. If this path is not unique, all paths that have made the same progress and are at the same position in the program, *equal progress paths*, are merged into one before the simulation is continued. The progress of a path is a record of how many times the simulation of that path has passed each loop header and entered each function, as well as how far the simulation has proceeded in the current loop iteration, as dictated by the program counter. The concept of path progress is refined further in the chapter about the implementation, Chapter 4.

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

2.4.3 Time-complexity and termination

For a program with a single feasible path, the analysis time will be proportional to the final WCET of the program. If the program contains several feasible paths, the time complexity depends on how many paths need to be simulated, how often merging is done and how fast the actual merge operation is. How often the merging should be done involves an important tradeoff. If merging is done too often, it is likely that there will be fewer paths to simulate. However, when doing a merge, information can be lost because variables whose values differ are assigned *unknown*. This may result in infeasible program paths not being eliminated which in turn increases the number of simulated paths. Thus, merging less often can actually lead to a fewer number of simulated paths.

The algorithm we have implemented merges as often as possible. This makes an exponential growth of the number of paths impossible, e.g. in loops. Typically, each unknown branch found during the simulation would

Algorithm 1 WCET algorithm handling merge.

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

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

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

wcet ← max_{p∈C} p.wcet
```


lead to the creation of yet another path, which later would result in an additional merge operation. The total number of paths simulated, as well as the number of merge operations, can in some cases grow in proportion to the number of loop iterations done in the program. One example is a loop with one unknown exit condition. The simulation of this loop would produce one new path (the exit path) to simulate each iteration. All these paths would, unless they reach the end of the program, be merged resulting in an equal number of merge operations.

For each merge operation, one must union the content of all registers and memory locations. This might be a quite slow process if the amount of memory is large. However, it is possible to speed up this operation considerably which we show in Chapter 4.

Another important issue is termination of the analysis. If the analysis encounters a loop that has an unknown exit condition it will not terminate. We have not implemented any solution to this problem but possible ways to handle it involve user interaction in some manner. For example, by providing constant feedback to a user about the progress of the analysis, the user can stop the analysis whenever the number of loop iterations seems too high. Another solution, that has been proposed by Gustafsson [Gus00], is to give a time budget before starting the analysis (for example a program deadline). If the WCET estimate exceeds this budget, the analysis can be stopped since the result is no longer interesting.

2.5 Discussion: value domains and annotations

Ideally, the goal is to eliminate all infeasible program paths. The domain of values we use, however, makes it sometimes impossible to correctly analyze mutually exclusive paths. Consider for example the statements in Figure 2.3a where b is *unknown*. Variable b will be *unknown* in both conditions, forcing the simulation of four paths even if `fun1()` and `fun2()` are mutually exclusive and the number of feasible paths are three. This can cause an overestimation of the WCET.

A more serious problem with our simple domain is that it can fail to convey information needed to terminate the analysis. If all exit conditions of a loop are input data dependent, we get a completely unknown upper bound on the number of iterations in the loop, and our WCET algorithm

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

(a)

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

(b)

Figure 2.3: Example of (a) mutually exclusive paths and (b) indirect loop bound.

will not terminate. For example, the loop in the program in Figure 2.3b, where b is unknown input data, will never do more than 100 iterations regardless of b. This fact cannot be represented with our simple domain, and during the simulation the loop will get an unknown exit condition. A similar problem can also arise when merging. The union operation used when merging may cause information needed to bound a loop to be lost.

Both problems above would be solved by using a more complex domain or by adding manual annotations that guide the analysis. Our approach does not hinder us from extending the domain of values to, e.g., intervals. However, our implementation of the method, which we evaluate in this thesis, uses the simpler domain. As we will se in the experimental results in Chapter 5, this simple domain performs remarkably well. Thus, for the purpose of obtaining tigth estimates it may be unnecessary to use a more complex domain, e.g., based on intervals of values [EG97]. On the other hand, the use of the simple domain forces us to add manual annotations in some cases. The implications of implementing another domain is discussed further in Chapter 4.

It is clear that the analysis sometimes need support from manual annotations. We distinguish between manual annotations that can affect the safeness of the derived WCET estimates and annotations that only guide the analysis. In our implementation, we have not added any support for annotations that influence the safeness. Instead we suggest using program modifications to add information in a safe way to a program. For example, to add information about the infeasible path in the program in Figure 2.3a, we can modify the program to explicitly carry the information that $b < 100$:


```
bsmall = 0
if (b < 100) {
    bsmall = 1;
    fun1()
  }
if (bsmall == 0 && b > 200)
  fun2()
```

To add an upper bound to the program in Figure 2.3b, we can add an extra condition that does not depend on input data:

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

When doing the experimental evaluation (see Chapter 5), we have only done modifications on loop bounds to add information necessary for termination. Information about infeasible paths have not been added.

The advantage with program modifications is that we will derive safe estimates. Manual annotations could introduce errors that jeopardize the estimated WCET. However, program modifications is not possible to use if the program cannot be changed (lack of source code for example) so manual annotations migth be needed as well. Ideally, we would like to permit as flexible annotations as described by, for example, Park's regular expression language [Par93] or the annotations possible in constraint solving approaches [LMW95, PS97]. However, it is currently unclear if this can be done and further research is needed to find out to what extent annotations can be introduces. Currently, the prototype tool only implements a rudimentary support for annotations that guide the analysis. See Chapter 4 for more information about this.

2.6 Related work

Previously presented methods that include automatic path analysis [Alt96, CBW94, EG97, FHL+01, HSR+00, HW99, GL02, NP93, EY97] are all based on similar concepts. To identify loop bounds and infeasible paths one must look at possible values of variables and find dependencies between variables and control flow instructions. This typically involves some kind of symbolic analysis of the program. However, despite the similarities, the methods still differ according to, for example, (1) the time

complexity of the analysis, (2) the expressiveness of the symbolic analysis used, and (3) whether the analysis is done for a high or low-level language. The time complexity of a method is typically high if all iterations in a loop are analyzed separately and low if the analysis is based on data flow analysis or some other technique that do not examine all iterations of a loop.

The symbolic execution method presented in this thesis has a high time complexity since the method analyzes all iterations of a loop. The domain used for the symbolic analysis is very simple since variables can only have a known value or be unknown. Furthermore, the analysis is done on the object-code (machine code) level. We will now compare this to other approaches.

Other methods with high time complexity typically use more expressive value domains. For example, the path analysis method proposed by Ermedahl and Gustafsson [EG97, Gus00] is similar to our method but their method works on the source code level instead of the object code level and uses a domain that can represent split integer intervals. Also, they focus only on deriving path information to be used later in a separate timing analysis. Compared to our approach, the more complex domain can potentially eliminate a greater number of infeasible paths and be more convenient to a user since constraints on the input data can be easily expressed. The language level issue is more tricky. On the source code level more high-level information is available and an analysis is more portable between different instruction set architectures. On the other hand, one can argue that on the object code level more low-level information is available and an analysis is more portable between different high-level languages.

Another method with high time complexity is the one presented by Altenbernd [Alt96]. It estimates the WCET by using a branch-and-bound algorithm and the symbolic evaluation is done on the source code level using a domain of value ranges. A similar symbolic evaluation is also used by Stappert and Altenbernd [SA00] when analyzing programs with no loops. There are also methods with high time complexity that are based on partial evaluation. Gómez and Liu [GL02] transform Scheme programs to time-bound functions. These rather complicated functions are then partially evaluated to produce more simple time-bound functions where infeasible paths have been eliminated. Nirkhe and Pugh [NP93] use partial evaluation to transform a program into a less complex one that

may contain fewer infeasible paths and simpler loop bounds.

Methods with low time complexity can be based on data-flow analysis, which typically works close to the object code level. This is the case for the method presented by Ferdinand et al. [FHL+01] which uses a domain of integer intervals to analyze the values of processor registers and identify infeasible paths. Healy et al. [HSR+00, HW99] present several algorithms that can derive upper bounds on loops and detect infeasible paths. By analyzing loop exit conditions, they can derive lower and upper bounds on loop iteration counters. To identify infeasible paths they build local arithmetic expression of each branch condition and then use a data-flow algorithm to find dependencies between registers and branch conditions and correlations between branches.

Chapman et al. [CBW94] combine program proof and WCET analysis. They use a symbolic domain where values can be arithmetic expressions to analyze programs written in the SPARK Ada language. A theorem prover is used to identify infeasible paths among a set of basic paths. A basic path represents at most a single iteration of a loop.

Finally, Ernst and Ye [EY97] use symbolic evaluation with arithmetic expressions to find the branch conditions that depend on input data. This is done to identify those parts of the program that have a single feasible path.

Some interesting conclusions can be drawn from a comparison of our method with other methods. First, all other methods use a more expressive symbolic analysis, based either on integer intervals or arithmetic expressions. Thus, other methods have the potential of identifying a greater number of infeasible paths. However, since our method analyzes all loop iterations, it can perform well despite the simple domain. In fact, the simple domain we use would be meaningless to use in an approach with low time complexity since loop iteration counters would become unknown, giving no information about the number of iterations done. Second, the choice of language to analyze varies greatly. Our main reason for analyzing object code instead of source code is the close connection to the timing analysis since the timing of instructions is typically more precisely defined at the object code level. The close connection to the timing analysis is one of the key ideas with our approach and makes it possible to do path and timing analysis simultaneously.

Another technique used to make the path analysis more flexible is to express the estimated WCET symbolically as a function of, for ex-

ample, the number of loop iterations. Several methods use this technique [CBW94, VHMW01, GL02, PB01, BB00]. There are also automatic path analysis methods that target synchronous languages [KW98] and hardware circuits [AS92]. A further discussion about infeasible paths can be found in [HFL01].

An important question is how to combine automatic path analysis with timing analysis. One approach is to do timing analysis before doing path analysis and do the WCET calculation as part of the path analysis [Alt96, GL02]. This can avoid the need of transferring path information to a separate WCET calculation step. However, it is not clear if this approach is powerful enough to cope with pipelined execution and cache memories. Another approach is to first derive path information which is then later used in the WCET calculation [CBW94, EG97, FHL+01, HSR+00, HW99, EY97]. This approach does not put any limits on the timing analysis but requires an explicit representation of the path information. The approach taken in this thesis avoids the need for an explicit representation of path information and still makes it possible to incorporate pipeline and cache analysis. In the next chapter, we will see how timing analysis is done and how it is integrated with the path analysis.

Chapter 3

Timing Analysis

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

In order to update the WCET properly during simulation, the simulator must of course be extended to model the timing of caches and pipelines. In the context of caches, the simulator must model the impact of cache misses on the execution time. And in the context of pipelines, the simulator must account for the impact of structural, data, and control hazards [HP96] on the execution time. With this capability, it is possible to make an arbitrarily accurate estimation of the WCET of a given path through the program.

A critical issue is how to carry out a merge operation. To do this, the method must estimate the impact of the system state on the future execution time. Such state information is exemplified by the identity of the blocks contained in the caches, which affects future misses, and the resources occupied by an instruction, such as data-path components and registers, which affect future structural and data hazards. The merge operation introduced in Section 2.4.1 must be extended to handle such state information, which we will refer to as *timing state*.

To merge the timing state, we could use the same general principle

as used when merging the content of memory locations; for all locations where the timing state differs we assert a pessimistic value, such as *unknown*, in the merged timing state. For example, in the case of caches, *unknown* can be similar to a cache block being invalid, and when calculating the union of two cache timing states, all cache blocks whose identities (i.e., memory tags) differ will be set to invalid. However, the introduction of abstract values, like the *unknown* value, can introduce extra complexity in the timing models. The basic merge algorithm described below works without extending the timing state with abstract values. However, in Section 3.5 we explain why abstract values like *unknown* can be useful and why we include them for instruction and data cache analysis.

In the next section, we will introduce the basic timing merge approach. We will concretely apply this method to a high-performance processor whose timing model is introduced in Section 3.2. We then explain in detail how the method is applied to model caching and pipelining in Sections 3.3 and 3.4, respectively. Finally, in Section 3.6, we discuss methods to reduce the possible pessimism introduced by the merge before we relate our timing analysis approach to others in Section 3.7.

3.1 Timing merge approach

Consider two paths p_A and p_B that are to be merged. A path p_A now consists of the functional state, $state_A$, which is the content of registers and memory, and the timing state, t_A, the state of the timing model which includes the estimated WCET so far. First, we assume that the content of memory locations and registers has been merged according to the principles stated in Section 2.4.1. Thus, we have created a new merged path, p_C, with a merged state, $state_C$, but still lacks the timing state, t_C. The remaining step is to merge the timing states, t_A and t_B, into t_C.

Our approach is based on the idea that if it was known in advance which of the paths p_A and p_B that belong to the worst-case path through the program, one could choose the timing state of that path when merging and discard the timing state of the other path. Then, no pessimism would be incurred on the final WCET estimate by the merge operation. The problem is how to identify the worst-case path. It is not enough to just compare the lengths, i.e., the current estimated WCET, of the two paths

due to the fact that the timing state in each path can differ and thereby cause different execution behavior in the future.

The solution is to not only compare the lengths of the two paths but also compare the timing states of the two paths. Then, if one of the paths is sufficiently longer than the other one, it can be guaranteed to be the worst-case path. If no clear decision can be made, it is always possible to force one of the paths to become the worst-case path by making it sufficiently longer. This is done by adding a *merge penalty* to the current estimated WCET. We will now take a more formal look on this solution.

As a starting point, we need to find out how the timing state t_A or t_B influence the final estimated WCET. We cannot know this before having analyzed the complete program. However, let us assume that when merging we set $t_C = t_A$ and continue simulating the program using the path p_C along *a single path* until the end of the program. The final estimated WCET will depend on the timing state t_A and the path taken through the rest of the program. Let us call the final estimated WCET, $wcet_{t_A}(x)$, where x is a parameter representing the path taken from the current location to the end of the program. Starting the simulation with t_B, we would obtain a similar final estimated WCET, $wcet_{t_B}(x)$. Figure 3.1 illustrates the concepts introduced so far.

We want to pick the timing state that leads to the greatest final WCET estimate. But, since we have not analyzed the whole program yet, we cannot know which path x that leads to the greatest estimated WCET. Furthermore, we have no knowledge of the absolute values of $wcet_{t_A}(x)$ or $wcet_{t_B}(x)$. Nevertheless, we do know something. By comparing the timing states t_A and t_B, we can estimate an upper bound on the maximum difference between the final WCET estimates. Let us define d_{AB} and d_{BA} as the maximum difference among all paths x:

$$d_{AB} = \max_x(wcet_{t_A}(x) - wcet_{t_B}(x))$$

$$d_{BA} = \max_x(wcet_{t_B}(x) - wcet_{t_A}(x))$$

The estimated upper bounds, which we will call $\Delta WCET(t_A, t_B)$ and $\Delta WCET(t_B, t_A)$, must fulfill:

$$\Delta WCET(t_A, t_B) \geq d_{AB}$$

$$\Delta WCET(t_B, t_A) \geq d_{BA}$$

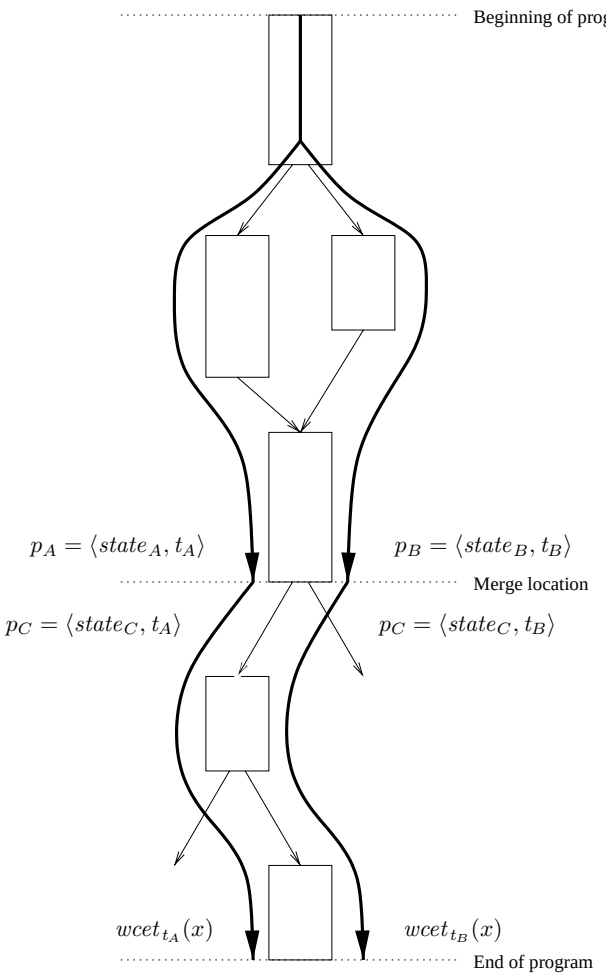


Figure 3.1: Two paths p_A and p_B are being merged. The new merged path, p_C, contains the merged functional state $state_C$. If we continue simulating p_C along a path x we get a final estimated WCET of $wcet_{t_A}$ or $wcet_{t_B}$ depending on which of the timing states, t_A or t_B, that is used as a starting point.

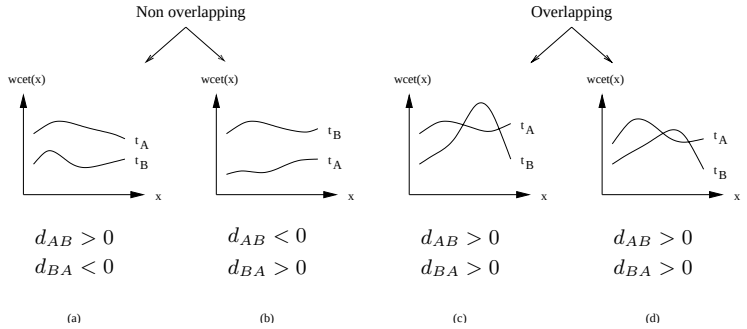


Figure 3.2: Four cases showing the final estimated WCET based on either t_A or t_B as a function of the path taken through the rest of the program, x.

How to really calculate such upper bounds on the future estimated WCET difference is the key to our timing analysis approach and is the central topic of the rest of this chapter. Basically, it is possible to estimate the difference by instead of considering all paths x that exist in the program from the current location to the end of the program, we extend the set of paths to consider all possible sequences of instructions of arbitrary length. Thus, the maximum difference is estimated by considering all possible combinations of future instruction sequences.

To better understand the meaning of these $\Delta WCET$ estimations, we will now take a look at Figure 3.2. It shows a graph of $wcet_{t_A}(x)$ and $wcet_{t_B}(x)$ for four different cases. In the first two cases, (a) and (b), one of the timing states will always lead to a greater estimated WCET regardless of which path x we choose. In these cases, either d_{AB} or d_{BA} will be negative. In the last two cases, (c) and (d), the curves overlap and both d_{AB} and d_{BA} will be positive. The difference in sign between d_{AB} and d_{BA} give us a possibility to identify the timing state that will result in the greatest final estimated WCET. This is done by also looking at the signs of the upper bounds $\Delta WCET(t_A, t_B)$ and $\Delta WCET(t_B, t_A)$. According to the signs of these upper bounds, we act according to the following decision table:

$\Delta WCET(t_A, t_B)$	$\Delta WCET(t_B, t_A)$		Action
> 0	≤ 0	$d_{BA} \leq 0$	Choose t_A. Discard t_B.
≤ 0	> 0	$d_{AB} \leq 0$	Choose t_B. Discard t_A.
$= 0$	$= 0$	Equivalent.	Choose either t_A or t_B.
> 0	> 0	Unsafe case.	d_{AB} and d_{BA} are possibly > 0. Force one timing state to become longer by adding a merge penalty to the current estimated WCET.

This means that we discard the timing state of the short path and continue the simulation using the timing state from the long path. The unsafe case is handled according to which difference is the greatest:

- If $\Delta WCET(t_A, t_B) > \Delta WCET(t_B, t_A)$, then t_A is forced to become longer by extending it with a merge penalty on the current estimated WCET. The penalty to add is $\Delta WCET(t_B, t_A)$. This penalty will increase $\Delta WCET(t_A, t_B)$ and decrease $\Delta WCET(t_B, t_A)$ so that we end up in a safe case according to the table above.
- If $\Delta WCET(t_B, t_A) > \Delta WCET(t_A, t_B)$, then t_B is forced to become the long path. The penalty to add is $\Delta WCET(t_A, t_B)$.

By adding a penalty we can compensate for the possible mistake we are doing when discarding one of the timing states. This will make the merge operation safe but will also introduce some pessimism in the final estimated WCET. However, this pessimism will only be incurred for the unsafe cases. When one path is clearly longer than the other one, no pessimism will be added to the final estimate.

In order to make this algorithm useful, we must be able to calculate the upper bound $\Delta WCET$. To simplify this calculation, we split the problem and treat each analysis of an architectural mechanism in separation. Thus, we split the timing state t into PL, IC, and DC to analyze the pipeline state, the instruction cache state, and the data cache state, respectively, and to analyze the system, we calculate:

$$\begin{aligned} \Delta WCET(t_A, t_B) &= \Delta_{pipeline}(PL_A, PL_B) + \\ &\quad \Delta_{IC}(IC_A, IC_B) + \end{aligned}$$

$$\Delta_{DC}(DC_A, DC_B) + pen_A - pen_B$$

where $\Delta_{pipeline}$, Δ_{IC}, and Δ_{DC} are the contributions to the upper bound regarding the timing state of the pipeline, the instruction cache and the data cache, respectively. The expression $pen_A - pen_B$ is also needed to account for the possible difference in the merge penalty accumulated from previous merges. A big contribution from $\Delta_{pipeline}$ means that the lengths of the paths differ considerably since the pipeline timing model is responsible for holding the current cycle count, i.e., the estimated WCET so far.

We will now demonstrate how to calculate the different contributions to the upper bound $\Delta WCET$. But first, we will present the hardware system used for the demonstration, a high-performance processor that uses many realistic features, and also define the timing state needed to simulate and analyze the system.

3.2 Modeled architecture

The architecture used to demonstrate the timing analysis can be seen in Figure 3.3. It consists of a multiple-issue pipeline, capable of in-order dispatch of two instructions each clock cycle, and separate instruction and data caches. While the architecture is a subset of the PowerPC instruction set architecture—floating-point instructions are excluded—it nevertheless contains many of the key features critical for high-performance processors such as pipelining and caching. We will now first describe the architecture and then present the timing model used when simulating the timing behavior.

3.2.1 Description

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

At most two instructions in each cycle can be fetched from the instruction cache and put into the buffers of the ID-stage. For simplicity,

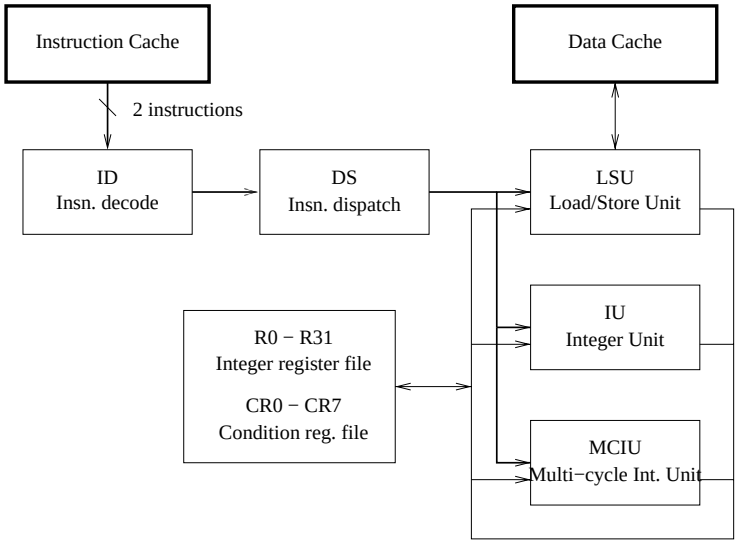


Figure 3.3: Modeled architecture.

we model no control hazards and assume that a branch is handled ideally by not incurring any penalty, i.e., instructions will be fetched from the correct branch target in zero cycles. Instruction fetching will stall if the buffers of the ID-stage are full or if the fetch causes an instruction cache miss.

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

All operands needed by an instruction are read from the register files

or forwarded from another functional unit at dispatch time. There is one result-bus from each unit handling the write-back of data into the register file and the forwarding of data to another unit.

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

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

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

3.2.2 Timing model

To simulate the timing behavior of this system we use a timing model consisting of three parts, the instruction cache, the data cache, and the pipeline. An example of the timing state needed to keep track of execution time can be seen in Figure 3.4. For each instruction, the functional behavior is first simulated, and then, the timing model is updated to account for the execution time.

A cache state is represented by an array of block identities, tags, showing how memory blocks are currently mapped in the cache or if a block is invalid (denoted by "X"). In the figure, the instruction cache, IC, is direct mapped and the data cache, DC, is 2-way set-associative. However, this is only an example. In this thesis, principles for cache

s	$IC(s)$
0	0
1	1
2	18
3	19
4	0
5	0
6	1
7	1
8	1
9	X
10	X
11	X
12	18
13	18
14	X
15	X

	$DC(s,i)$	
s	$i=0$	$i=1$
0	0	1
1	1	X
2	18	X
3	19	X
4	0	18
5	0	18
6	1	X
7	X	X

Resource r	Release time $PL(r)$
ID0	120
ID1	120
DS0	121
DS1	121
LSU	123
IU	122
MCIU	124
R0	124
R1	122
R2	123
R3	60
...	
R31	100
CR0	109
CR1	109
...	
CR7	109

Figure 3.4: Examples of instruction cache, IC, data cache, DC, and pipeline, PL, timing states. A cache state is an array of block tags where a tag of "X" means that the block is invalid. For set-associative caches, $i = 0$ represents the youngest block in a cache set. The pipeline state is a reservation table showing the latest release times for each pipeline resource.

configurations with arbitrary size and associativity will be covered.

The following notation will be used for referring to the tags in an instruction cache or a data cache. For an m-way set-associative cache with n sets, the timing state is represented by a 2-dimensional array, $IC(s,i)$ or $DC(s,i)$, with size $n \times m$. The tag $IC(s,i)$ or $DC(s,i)$ is then tag number i in set s. For an LRU replacement policy, i.e., the least recently used block is replaced when a cache miss occurs, the ordering of tags in a set reflects their relative LRU status, i.e., tag $i = 0$ belongs to the most recently used block and $i = m - 1$ to the least recently used block, which is the one to replace next. To refer to all tags in a cache set, we will use the notation $IC(s)$ or $DC(s)$. For direct mapped caches, we will skip the last index and simply use $IC(s)$ or $DC(s)$.

The pipeline state is responsible for modeling the penalties incurred by resource contention (structural hazards) and register dependencies (data hazards). For our pipeline model, resource contention and dependencies can be modeled using a pipeline reservation table, which records when each resource (pipeline stage or register) is released. An example of such a table can be seen in Figure 3.4. The ID- and DS-stages are divided into two sub-stages, ID0, ID1 and DS0, DS1, since each stage can hold two instructions at a time and each pipeline-stage buffer is treated as an individual resource. If two instructions are present in the pipeline stage, then both sub-stages are reserved. If only one instruction is present, then only ID1 (or DS1) is reserved.

During simulation, the reservation table can be updated for each instruction at a time because all resources that the instruction requires are known. First, the instruction and data cache accesses are simulated. Then, the pipeline reservation table is updated to show when each resource is released. The current estimated WCET of the path is defined as the time when the last resource is released in order to make a safe estimation. However, the accumulated merge penalty, i.e., the extra cycles added in some cases to make a safe merge, must also be added. Thus, the estimated WCET can be expressed as:

$$\text{estimated } wcet = \max_{r \in R}(PL(r)) + pen$$

where R is the set of all pipeline resources and pen is the merge penalty accumulated from earlier merges.

3.2.3 Handling unknown accesses

During estimation, the reference address of a memory access can become *unknown* due to dependencies with unknown input data. Without knowledge of the reference address, we cannot update the state of the data cache properly. This can be solved in several ways. One solution is to let the state of the data cache remain unchanged and add two cache miss penalties [KMH96] — one for the possible miss and one for the possible replacement of some useful memory block. However, in line with our handling of unknown store instructions in Section 2.3 where we map unpredictable data structures to a certain region in memory, we can instead use the approach of disabling caching for this memory region. Then, only one cache miss penalty needs to be added. We develop this approach further in Chapter 7.

3.3 Instruction and data cache analysis

In the following, we describe how to calculate the contribution from the instruction and data cache, Δ_{IC} and Δ_{DC}, to the upper bound $\Delta WCET$ as introduced in Section 3.1. Both instruction and data caches can be described and treated using the same principles and we use the instruction cache to demonstrate the principles. We will begin showing how to analyze direct-mapped caches and then extend the method to handle set-associative caches, focusing on the least-recently used (LRU) replacement strategy.

3.3.1 Direct-mapped cache

Let IC_A and IC_B be the timing state of the instruction caches corresponding to two paths p_A and p_B, respectively. To calculate the upper bound $\Delta_{IC}(IC_A, IC_B)$, we must consider all cases where IC_A may lead to a greater number of future cache misses than IC_B would. The worst case is found if we imagine that all cache blocks resident in IC_B but not found in IC_A will be needed in the future. Accesses to these blocks would result in cache misses in IC_A but cache hits in IC_B. To find the total number of extra cache misses possible from IC_A compared to IC_B, we go through all cache blocks and compare the tag in IC_A with the tag in IC_B and determine the number of entries where they differ. This number

is then multiplied by the cache miss penalty to form the contribution to the upper bound Δ_{IC}.

Formally, Δ_{IC} for a direct-mapped instruction cache can be expressed as:

$$\Delta_{IC}(IC_A, IC_B) = P_{IC} \sum_{s=0}^{n-1} c(s)$$

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

$$c(s) = \begin{cases} 1 & \text{if } IC_A(s) \neq IC_B(s) \text{ and } IC_B(s) \neq \text{invalid} \\ 0 & \text{otherwise} \end{cases}$$

where $IC_A(s)$ and $IC_B(s)$ denote the cache tag for the block in set s. We get a contribution if the tags differ between cache blocks in the two cache states, but not if the cache block in IC_B is invalid. Then, it is impossible for IC_A to cause any additional misses in the future.

3.3.2 Set-associative cache

To derive the upper bound Δ_{IC} for set-associative caches, we consider again the timing state of two instruction caches, IC_A and IC_B, that are to be merged. We will begin by constructing a pessimistic expression for Δ_{IC} and then refine it based on the characteristics of the LRU replacement algorithm.

We want to estimate the number of extra cache misses possible from IC_A compared to IC_B. A pessimistic expression can be obtained by extending the expression for direct-mapped caches in a straight-forward way. Instead of making the comparison for only one tag in the set, we now must consider all tags in a set. If any tag in a set is found to differ between IC_A and IC_B we assume that this can make all tags for blocks belonging to the same set differ. Thus, for an m-way set-associative instruction cache with n sets, Δ_{IC} can be expressed quite similarly to the direct-mapped case:

$$\text{Pessimistic } \Delta_{IC}(IC_A, IC_B) = P_{IC} \sum_{s=0}^{n-1} mc'(s)$$

i	0	1	2	3	4	5	6	7
$IC_A(s,i)$	10	11	12	14	4	15	16	17
$IC_B(s,i)$	10	11	12	3	14	15	16	x

Figure 3.5: Example of cache states for one set s in an 8-way cache. A tag "x" means that the block is invalid. $i = 0$ is the most recently used block.

where $c'(s)$ now indicates if sets differ with respect to at least one tag as defined by:

$$c'(s) = \begin{cases} 1 & \text{if } IC_A(s,i) \neq IC_B(s,i) \text{ and } IC_B(s,i) \neq \text{invalid, for any } i \\ 0 & \text{otherwise} \end{cases}$$

This expression is often overly pessimistic and in the rest of this section we will show how to make a more accurate calculation. To begin, let us study the example in Figure 3.5. This figure shows an example cache state for a single set in an 8-way associative cache. In the example, $IC_A(s,i) \neq IC_B(s,i)$ for both $i = 3$ and $i = 4$. This makes $c(s) = 1$ and the pessimistic expression above would count this as 8 possible extra cache misses. However, this is quite pessimistic. In reality, only 3 extra cache misses can occur from IC_A compared to IC_B. To see how this is possible, we will now first present an algorithm that calculates the number of extra misses by constructing a worst-case access pattern. Then, guided by a proof of this algorithm we will construct an alternative expression to calculate the number of extra cache misses, which is the one we use in our implementation.

A *worst-case access pattern* is a list of future accesses targeting a specific cache set s that causes the maximum number of misses for cache set s in IC_A, compared to the same cache set in IC_B. All accesses in the list refer to distinct memory blocks and the algorithm we will present finds one example of such an access list. The constructed list consists of at most m accesses since after m unique accesses both cache states, $IC_A(s)$ and $IC_B(s)$, will become identical.

The algorithm is presented in Figure 3.6. In each iteration an access is added to the list and the effect on the cache state of making this access is simulated. However, the original cache state must not be destroyed. Therefore, the update is done to S_A and S_B, which are initialized as copies

```
# Construct worst-case access list, l, for set s.
# x(s) is the number of extra misses for this cache set.

S_A = IC_A(s)
S_B = IC_B(s)
x(s) = 0
while S_A ≠ S_B
  a = a block in S_B that do not exist in S_A
  if a block a was found then
    add access targeting a to l
    update S_A and S_B with new access
    increase x(s) by 1
  else
    add arbitrary access targeting a new block to l
    update S_A and S_B with new access
    # (will cause a miss for both S_A and S_B)
  end for
end while
```

Figure 3.6: Worst-case access pattern algorithm for LRU replacement.

of IC_A and IC_B. In Figure 3.7 the algorithm is applied to the example in Figure 3.5. In iteration 1, S_B (the copy of IC_B) is found to have a block with tag 3 that does not exist in S_A. An access to this memory block is added to the access list and the number of extra misses is incremented since this access would miss according to S_A but not according to S_B. In the next iteration, the cache states have been updated with this new access with tag 3, and no more blocks are found that only exist in S_B and not in S_A. Thus, an arbitrary access is added that does not incur any extra miss. When the cache states are updated with this new access, the oldest block in each cache state is replaced. This makes it possible, in iteration 3, to add the block with tag 16, which has been replaced from S_A, to the access list. The algorithm continues for one iteration more, adding the block with tag 15 to the access list since this block was replaced in S_A in the previous iteration. Finally, the cache states are equal and the algorithm terminates. The total number of extra misses that $IC_A(s)$ can cause compared to $IC_B(s)$ is found to be 3.

Theorem 3.1 *The access pattern created by the algorithm in Figure 3.6 is a worst-case pattern.*

Informal proof Being a worst-case pattern means that no other sequence of accesses can make the number of extra misses from $IC_A(s)$ compared to $IC_B(s)$ to become greater. To prove this we look at all blocks present in $IC_B(s)$. To count as an extra miss, an access must reference a block present in $IC_B(s)$ and not in $IC_A(s)$. This can be accomplished in two ways:

1. An access to a block in $IC_B(s)$ that does not exist in $IC_A(s)$ must clearly count as an extra miss and also be included in the worst-case pattern. This case is handled by the algorithm in the *then* clause of the if statement.

2. An access to a block in $IC_B(s)$ that also exists in $IC_A(s)$ can be made to cause an extra miss if the block can be replaced earlier in $IC_A(s)$ than in $IC_B(s)$. This case is discovered by the algorithm by forcing a replacement of one block in each iteration. The *else* clause guarantees that a block is being replaced in each iteration by adding an arbitrary new access. Furthermore, if a block is being replaced in $IC_A(s)$ but is still left in $IC_B(s)$, the *then* clause will discover this and count it as an extra miss.

i	0	1	2	3	4	5	6	7
$S_A(i)$	10	11	12	14	4	15	16	17
$S_B(i)$	10	11	12	3	14	15	16	x

1. Access to block 3 is added. Access list: {3}. $x(s) = 1$.

i	0	1	2	3	4	5	6	7
$S_A(i)$	3	10	11	12	14	4	15	16
$S_B(i)$	3	10	11	12	14	15	16	x

2. Arbitrary access added. Access list: {3, a}. $x(s) = 1$.

i	0	1	2	3	4	5	6	7
$S_A(i)$	a	3	10	11	12	14	4	15
$S_B(i)$	a	3	10	11	12	14	15	16

3. Access to block 16 added. Access list: {3, a, 16}. $x(s) = 2$.

i	0	1	2	3	4	5	6	7
$S_A(i)$	16	a	3	10	11	12	14	4
$S_B(i)$	16	a	3	10	11	12	14	15

4. Access to block 15 added. Access list: {3, a, 16, 15}. $x(s) = 3$.

i	0	1	2	3	4	5	6	7
$S_A(i)$	15	16	a	3	10	11	12	14
$S_B(i)$	15	16	a	3	10	11	12	14

$S_A = S_B \Rightarrow$ stop. Final list: {3, a, 16, 15}. Total number of extra misses: $x(s) = 3$.

Figure 3.7: Applying the worst-case access pattern algorithm to the example in Figure 3.5. In the list, only the tag-part of an address is included.

Thus, the algorithm finds the worst-case number of extra misses possible from $IC_A(s)$ when compared to $IC_B(s)$. ◇

The worst-case pattern algorithm can be used to calculate an upper bound Δ_{IC} that is as tight as possible by calculating the number of extra misses, $x(s)$, for each cache set. The upper bound is given by:

$$\Delta_{IC}(IC_A, IC_B) = P_{IC} \sum_{s=0}^{n-1} x(s)$$

Before we end this section, we will present an alternative expression to calculate the upper bound. This expression is the one we have based our implemention on and can be more convenient and efficient to implement since it does not need to update any cache state.

The proof above can be translated to an expression that tells us if a block in IC_B will give rise to an extra miss. The key observation needed is that a block, b, in IC_B, can be replaced earlier in IC_A than in IC_B if there exists another younger block in IC_A that does not exist among the blocks younger than b in IC_B. In other words, all blocks younger than b in IC_A must also exist in IC_B and be younger than b in IC_B, in order for block b to survive in the worst case. Formally, the upper bound Δ_{IC}, can now be expressed as:

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

where $c(s,i)$ is defined by:

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

An example of applying this expression to the previous cache state example from Figure 3.5 can be found in Figure 3.8. The expression $c(s,i)$ will be 1 for all blocks in $IC_B(s)$ that will cause extra cache misses

i	0	1	2	3	4	5	6	7
$IC_A(s,i)$	10	11	12	14	4	15	16	17
$IC_B(s,i)$	10	11	12	3	14	15	16	x
$c(s,i)$	0	0	0	1	0	1	1	0

Figure 3.8: Applying the alternative expression to the example in Figure 3.5.

when accessed, i.e., block 3, 15, and 16, which are the same blocks as found by the algorithm above. Block 3 only exists in $IC_B(S)$ and block 15 and 16 can be replaced earlier in $IC_A(s)$ than in $IC_B(s)$ due to block 4 in $IC_A(s)$. However, this alternative expression gives no information about in which order to access the blocks in order to trigger this worst-case scenario.

The alternative expression is the basis for our implementation. However, we have improved it further and the final algorithm is presented in the implementation chapter, Chapter 4.

3.4 Pipeline analysis

To finally be able to calculate the upper bound $\Delta WCET$ as introduced in Section 3.1, we need also to define how to calculate the contribution from the pipeline state, $\Delta_{pipeline}$.

Consider the timing state of the pipelines, PL_A and PL_B, corresponding to the two paths p_A and p_B, respectively (see the example in Figure 3.4). We want to find an upper bound, $\Delta_{pipeline}$, on the difference in the final estimated WCET that PL_A can lead to compared to PL_B, and must in this case consider all future hazards that PL_A can lead to which PL_B cannot lead to. Hazards are taken care of by updating the release time of a resource to indicate when it will be available. Therefore, by examining each resource and compare the release times in PL_A and PL_B, we can find out how much each resource can influence the future execution. Using this, we can construct an upper bound for the pipeline by picking the maximum difference found. This can be expressed as follows, where R is the set of all resources in the pipeline reservation table and $PL_A(r)$ and $PL_B(r)$ are the release times for resource r:

Resource	Release time		
r	$PL_A(r)$	$PL_B(r)$	$PL_A(r) - PL_B(r)$
ID0	120	100	20
ID1	120	100	20
DS0	121	101	20
DS1	121	101	20
LSU	123	113	10
IU	122	102	20
MCIU	124	94	30
R0	124	94	30
R1	122	102	20
R2	123	113	10
R3	94	99	-5
...			
R31	100	100	0
CR0	109	89	20
CR1	109	89	20
...			
CR7	109	89	20

Figure 3.9: Example of two pipeline states and the difference in release times for each resource.

$$\text{Pessimistic } \Delta_{pipeline}(PL_A, PL_B) = \max_{r \in R}(PL_A(r) - PL_B(r))$$

We call this the pessimistic $\Delta_{pipeline}$ since by only looking at the release times we can get a very pessimistic upper bound. As an example, let us compare the two pipeline states presented in Figure 3.9. For this example, we get $\Delta_{pipeline}(PL_A, PL_B) = 30$ due to the difference in release times for resources MCIU and R0. This is overly pessimistic since, for PL_B, the earliest cycle in which a future instruction can use these resources is 103 and not 95 as the release time would suggest. This is explained by the fact that a future instruction must use the ID and DS resources before being able to use the MCIU and R0 resources. Therefore, we really want to look at the earliest use time for each resource that is possible for one or several future instructions. The earliest use time for a resource r can be calculated by taking the maximum of the release times of the resource r and the resources in previous stages of the pipeline. The pessimistic expression above can now be improved by instead of $PL_A(r)$

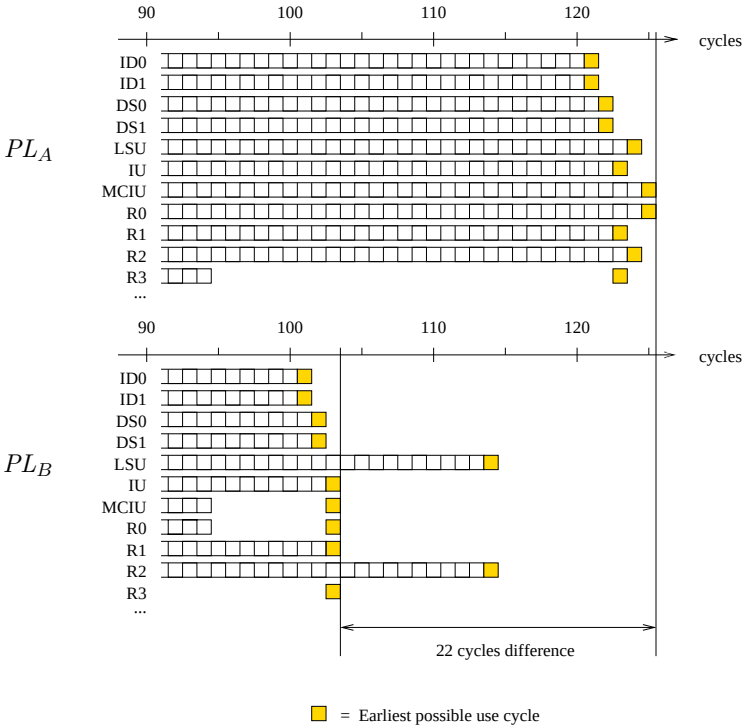


Figure 3.10: Illustration of earliest use times for the example in Figure 3.9. The earliest use times for the functional units and the registers are at least one cycle later than the earliest use time for the DS-stages.

and $PL_B(r)$ use the earliest use times, $u_A(r)$ and $u_B(r)$. We get:

$$\Delta_{pipeline}(PL_A, PL_B) = \max_{r \in R}(u_A(r) - u_B(r))$$

Continuing with the example, Figure 3.10 illustrates how the earliest use times are calculated for the two pipeline states PL_A and PL_B. Based on the difference in earliest use time, we now get: $\Delta_{pipeline}(PL_A, PL_B) = 22$.

3.5 Abstraction

Our method of merging timing states does not require any abstract state values, like for example the *unknown* used for memory and register values in Chapter 2. However, using no abstraction can cause great over-estimation of the WCET. This is the case when trying to solve the *initial state* problem, i.e., what initial timing state should we assume when doing a WCET estimation?

Without the use of abstractions, we can simply choose an arbitrary initial timing state, t_i, and then use this as a starting point for our simulations. The estimated WCET we derive is valid based on this initial timing state and we call it $wcet_{t_i}$. Assume now that after doing a WCET estimation, we want to use the estimate in a context where the initial timing state is not t_i but instead t_j. Then, we can safely calculate a new WCET estimate by pretending that we have an initial merge point that merges t_i and t_j but always add a merge penalty to t_i. Thus, the new WCET estimate, $wcet_{t_j}$, can be obtained by:

$$wcet_{t_j} = wcet_{t_i} + \Delta WCET(t_j, t_i)$$

The reasoning above can be generalized to calculate a WCET estimate that is valid for all possible initial states. This is what we generally want to calculate and by considering all possible timing states t_j, it can be obtained as:

$$wcet = wcet_{t_i} + \max_{t_j}(\Delta WCET(t_j, t_i))$$

Now, depending on the initial timing state t_i, the over-estimation can be more or less severe. We will now see how the use of abstraction can reduce this over-estimation.

Consider the initial timing state of an instruction cache, IC_i. If this initial cache contains specific memory blocks, then the penalty $\Delta_{IC}(IC_j, IC_i)$ can be high for an arbitrary IC_j. However, if IC_i only contains invalid blocks then $\Delta_{IC}(IC_j, IC_i) = 0$ for all IC_j (according to the definition of Δ_{IC} in Section 3.3). The invalid state in a cache represents the worst-case state in our cache model. However, in more complex cache models (for example, using the copy-back strategy), the worst-case can be something other than invalid. So, to be more general

we actually want to introduce a state, *unknown*, for a cache block that represents this worst-case. The state *unknown* should be defined so that $\Delta_{IC}(IC_j, IC_i) = 0$ for all IC_j where IC_i is *unknown* for all blocks.

To summarize, by setting the initial state to *unknown* values, we can eliminate the added penalty and directly derive an estimated WCET valid for all possible initial states. On the other hand, by setting the initial state to *unknown*, the estimated WCET can increase. For example, an invalid cache state will probably result in more cache misses when compared to many other initial cache state. Thus, abstraction is only meaningful when the typical increase in final estimated WCET is smaller than the eliminated penalty.

We use abstraction in the instruction and data cache timing models but not in the pipeline model. For cache memories, abstraction is a gain whenever the analyzed program does not use the whole cache, which is often the case. However, for pipeline models, the execution will typically influence all timing state. This makes it meaningless to introduce abstract state values. For our cache model, we assume that an invalid cache state is the worst-case. This makes it unnecessary to introduce any extra state bits in the cache model. Also, we always assume that the initial pipeline state is a flushed pipeline. This is a safe assumption since we define the estimated WCET to be the last release time among all pipeline resources.

3.6 Reducing the worst-case penalty

We have now seen that the merge operation can be used to produce a safe estimated WCET. A remaining question is how tight the estimate will be. This depends both on the timing model and the merge operation. The timing model can cause over-estimation by pessimistic assumptions, for example, assuming the highest latency for a variable-latency instruction when input data is unknown. However, this over-estimation can not easily be reduced. The merge operation can cause over-estimation by adding a merge penalty to the final estimated WCET. This penalty can typically become high if a program contains multiple paths to merge and the timing states of the paths are very different to each other but yet being equal in length. The experimental results in Chapter 5 show that the merge operation performs well for the programs that have been studied. Nevertheless, we will now present two methods, *speculative modifications*

and *delayed merge*, that can reduce the merge penalty. These methods have not been implemented since the need did not arise. However, it is fairly easy to construct program examples where the methods would be useful.

3.6.1 Speculative modifications

The method of speculative modifications involves making modifications to the timing state of the path chosen as the long path when merging, in a way that promises to reduce the final estimated WCET. The idea is based on the fact that when a merge penalty is to be added we can permit an equivalent amount of changes to the timing state.

As an example, let us consider a loop containing two paths p_A and p_B that are being merged in each iteration and where the merge causes a substantial penalty to be added. Assume that in one of the paths p_B an access, A, with cache tag a is made that affects cache set s in a direct-mapped data cache. This cache set is not accessed anywhere else in the program and no other accesses is made in the loop. Consider now the data cache states in the two paths that are to be merged, DC_A and DC_B, and the pipeline states, PL_A and PL_B. Assume that path p_A is chosen as the long path and that the total penalty, which is assumed to be the sum of Δ_{DC} and $\Delta_{pipeline}$, is calculated as follows in each iteration:

cache access A	miss	hit
$\Delta_{DC}(DC_B, DC_A)$	0	0
$\Delta_{pipeline}(PL_B, PL_A)$	10	0
Total penalty	10	0

Thus, if access A causes a cache miss, a penalty of 10 cycles must be added. The unfortunate case here is that access A will cause a miss in each iteration since the cache set vill always be invalid when the access is made. This is because the cache state in the long path $DC_A(s) = X$ will be kept and used in the next iteration. The penalty incurred can be relatively high if the paths are short.

To reduce the penalty, we can do a speculative change of $DC_A(s)$ from being invalid, "X", to being equal to $DC_B(s) = a$. If this modification is done after the first iteration, access A will experience a cache hit in each of the following iterations and no penalty will accumulate. The modification can be done safely by adding a penalty of 10 cycles to p_A.

However, the added merge penalty will then be reduced by 10 cycles so the total penalty added for the first iteration is the same as before the modification.

In the case above, the modification to do was quite easy to figure out. Just check all blocks that differ between DC_A and DC_B. If a block $DC_A(s)$ is invalid and $DC_B(s)$ is not, then $DC_A(s)$ should be modified. However, for a more general approach we must be able to detect changes in the timing state. Then, if the state $DC_B(s)$ has changed but not $DC_A(s)$ since the two paths p_A and p_B split up, we should modify $DC_A(s)$. Tracking changes is quite simple to do and in our implementation we already have preliminary support for this due to the fact that we keep track of changed memory content to speed up the merging (see Chapter 4).

3.6.2 Delayed merge

Another way of reducing the merge penalty is to use delayed merge. This means that we delay the merging and continue simulating past an unknown conditional branch until the next unknown conditional branch. In this way, the timing states are made more similar which can lead to a reduced penalty. The price we pay is an increased number of paths to simulate and an increased number of merge operations.

The principle is illustrated in Figure 3.11. By delaying the merge, the number of paths are allowed to increase exponentially. For example, by delaying the merge by 2 unknown conditional branches we must simulate 4 times as many paths and do 4 times as many merges compared to when doing an non-delayed merge. The key to the succes of this method is the way paths to be merged are chosen. For example, consider the paths to be merged when delay = 2 in Figure 3.11. We choose to merge two paths p_A and p_B, where p_A and p_B have followed the non-taken and taken branch target of branch 0, respectively, and where both paths have followed the same branch targets of branch 1 and 2. After branch 1, the timing states in both paths are being updated with the same instruction execution sequence. This makes the timing states in the two paths to become more similar to each other which will reduce the merge penalty.

The delayed merge has not been implemented but it would only require a relatively minor change of the WCET algorithm presented in Section 2.4.2. Also, no evaluation has been done to see how much the

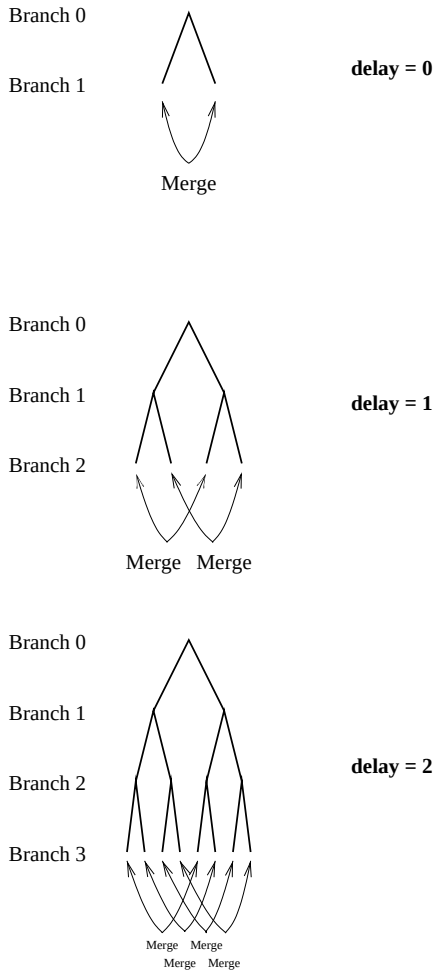


Figure 3.11: Delayed merge. Instead of merging directly at every unknown conditional branch, the paths are allowed to multiply. Then, paths with similar recent history are merged.

merge penalty can be reduced using this method.

3.7 Related work

The main difference with the WCET estimation approach presented in this thesis compared to other approaches is the strong focus on doing accurate timing analysis. This accurate timing analysis is accomplished by a close integration of path and timing analysis but also by the analysis of all loop iterations in a loop. The largest gain from an analysis of all iterations is related to data cache analysis and we will defer that topic to Chapter 7. In this section, we will demonstrate the gains possible from an integrated analysis and compare with other methods.

As a basis for our comparison, we will use the example program shown in Figure 3.12. It consists of a loop containing two if-statements, one with a known condition $a > 0$ and one with an unknown condition $b > 0$. Each block in the if-statements contains a memory access instruction. Also, the memory accesses made in the blocks at line 6 and 8 are assumed to cause a conflict with each other in the asummed direct-mapped cache. The same is assumed for the accesses made at line 11 and 13. One of the two possible paths in each if-statement is always longer than the other path even if a cache miss occurs when accessing memory.

For this example, an integrated approach will estimate the WCET as explained in Figure 3.13 (we assume that only blocks A,B,C, and D contribute to the WCET). Thus, an integrated approach can really find the actual WCET. Non-integrated approaches can overestimate the WCET due to mainly two reasons related to cache analysis. The first reason is due to the infeasible path present in the first if-statement—the else clause will never be executed. Information about infeasible paths is not always taken into account when doing cache analysis. The other reason is due to a separation of the cache analysis from the calculation of the WCET. We will now examine to what extent other approaches can handle the example program.

3.7.1 Constraint solving approaches

One popular approach used by many research groups is to use constraint solving techniques to estimate the WCET. The WCET is estimated by maximizing an algebraic expression representing the execution time under


```
 1  a = 1
 2  b = unknown input
 3
 4  for (i = 0 ; i < 1000 ; i++) {
 5    if (a > 0)
 6      A (accesses memory at 0x0500, cache set 5, cycles: 20/30)
 7    else
 8      B (accesses memory at 0x2500, cache set 5, cycles: 40/50)
 9
10    if (b > 0)
11      C (accesses memory at 0x0800, cache set 8, cycles: 20/30)
12    else
13      D (accesses memory at 0x2800, cache set 8, cycles: 40/50)
14  }
```

Figure 3.12: Example program that profits from an integrated analysis. The execution time for each block in the if-statements is expressed as X/Y, meaning the number of cycles when the memory access causes a cache hit or miss, respectively.

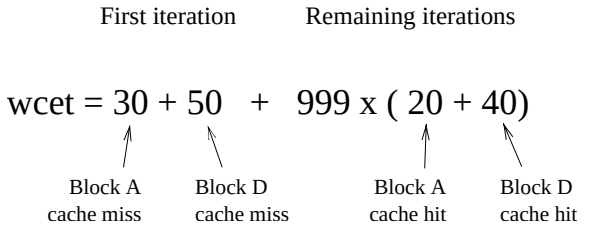


Figure 3.13: The WCET for the example program in Figure 3.12. Cache misses are assumed for the first iteration.

a set of constraints. The constraints provide a flexible way to take into account path information from a separate automatic path analysis [EE00] and we will in this discussion assume that such a path analysis is included. One of the first examples of this approach is the method presented by Li et al. [LMW95, LMW96]. They add cache analysis by including cache miss penalties into the execution time expression and by adding constraints that model conflicting accesses to the cache. The way they add cache analysis actually makes this into an integrated approach and their method would also find the actual WCET for our example program. The same is true for the cache analysis approach presented by Ottosson and Sjödin [OS97]. They extend a basic constraint solving approach [PS97] with pipeline and cache analysis. Instead of adding extra constraints to model cache conflicts, they extend the execution time expression to include effects over consecutive basic blocks[1] in the program.

While both of these constraint solving approaches perform well for our example program, they can suffer from complexity problems. For some fairly simple programs and normal cache architectures, analysis times are reported to be in the order of several minutes [OS97] or several hours [LMW96]. Although better constraint solving techniques can improve this situation, these approaches are still quite complex to use. To model an architectural mechanism, one must be able to convert its timing behavior into an algebraic expression. It is not really clear how easy these approaches can be extended if one wants to model a more detailed timing behavior. The complexity issue for an integrated constraint solving technique is probably the main reason that several other methods have been proposed that separate the cache analysis from the constraint solving phase. For example, Engblom [Eng02] further extends the approach of Ottosson and Sjödin [OS97] to handle pipeline effects over an arbitrary number of consecutive basic blocks. He suggests that cache analysis should be done separately to avoid the risk of a too long analysis time.

Several methods include cache analysis as a separate phase. The cache analysis presented by Ferdinand et al. [FW99] is based on data flow analysis (abstract interpretation). For our example program, this analysis would result in cache misses for the blocks A and D in all iterations, thus

[1] A basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [ASU86].

overestimating WCET by a factor of: $(30 + 50)/(20 + 40) = 1.33$. There are two reasons for this: (1) the cache analysis does not take into account information about the infeasible path and therefore must assume that the accesses from blocks A and B conflict, and (2) the cache analysis is done before we find out that block C never can be part of the worst-case path through the program. Thus, accesses from blocks C and D must be assumed to conflict as well. This is an example of a completely separated cache analysis. An interesting hybrid approach has been presented by Wolf and Ernst [WE00]. They do a kind of data flow analysis where the unit of analysis is extended from basic blocks to larger segments of the program that contain only a single feasible path. For our example program, their method would probably be able to treat the first if-statement as a single feasible path segment and therefore would treat the access from block A as a cache hit during all but the first iteration. However, the cache analysis seems to be separated from the WCET calculation so the access from block D must be assumed to be a cache miss. The overestimation will then become a factor of: $(20 + 50)/(20 + 40) = 1.17$.

The methods by Ferdinand et al. [FW99] and Wolf and Ernst [WE00] serve as good examples of the fact that the degree of separation can differ between different cache analyses. Recently, Ferdinand et al. [FHL+01] combine cache analysis with pipeline analysis in order to analyze the Motorola ColdFire microprocessor. However, the degree of integration is not really clear and it is hard to tell if they really do an integrated analysis or not.

3.7.2 Other approaches

There are numerous other approaches that are not based on constraint solving. Again, we can find examples of more or less integrated methods. The method proposed by Lim et al. [LBJ+94] and Kim et al. [KMH96] is an example of an integrated method that adds cache and pipeline analysis to the timing schema approach presented by Shaw [Sha89]. They do cache and pipeline analyses for each basic block in a program and then concatenate the results from each basic block into timing information regarding longer paths. Inside loops, they use a prune operation to discard paths that are guaranteed to be shorter. As presented, the method does not include any possibility of incorporating path information. However, we can probably assume that this can be included at least by means

of manual annotations. Thus, for our example program, the method would correctly assume a cache hit for the blocks A and D, resulting in no overestimation. Still, the approach can suffer from inaccurate timing analysis because of the concatenation approach. When concatenating the timing behavior of two basic blocks, one will always run the risk of loosing precision compared to when simulating the timing behavior of the two basic blocks.

The approach presented by Healy et al. [HWH95] and White et al. [WMH+97] is one of the first examples of a separated approach for cache analysis. They use static cache simulation [Mue00] to classify cache accesses. This is the technique that inspired Ferdinand et al. [FW99] when constructing their cache analysis. Thus, the overestimation: 1.33 is the same, representing a complete separation of path analysis, cache analysis, and WCET calculation. We assume that the automatic path analysis presented by Healy and Whalley [HW99] succeeds to find the infeasible path in the first if statement.

Another example of a separated approach is the one presented by Colin and Puaut [CP01, CP00]. Besides pipeline and cache analysis, they also include a branch prediction analysis. All analyses are done separately and then later combined to calculate the WCET. The infeasible path in our example program can be eliminated by a manual annotations. However, since the analyses are done separately, we will still get an overestimation of 1.33 as previously explained. Finally, the method proposed by Stappert and Altenbernd [SA00] is also a separated approach. They only analyze straight-line code without any loops but include an automatic path analysis. The overestimation will be 1.33 for this method as well.

To sum up, the WCET estimation method presented in this thesis is one example of an integrated analysis. However, when regarding timing analysis, it has the potential of doing a more accurate analysis than other methods. In Chapter 5, we will evaluate the path and timing accuracy of the symbolic execution method based on the prototype tool that will be described in the next section.

Chapter 4

Implementation

In this chapter, we will describe how we have implemented the WCET algorithm presented in Chapter 2 and the timing merge operation presented in Chapter 3. The tool we have constructed has been used to experimentally evaluate our ideas and the results from this evaluation is presented in the next chapter. The main goal when implementing the WCET method has been to make a tool that proves that the principles presented in this thesis can be feasibly implemented. This means that many aspects are lacking. For example, tool performance issues have only been given moderate consideration, the user interface is lacking, and the tool is not very portable to other platforms. Nevertheless, it provides a good example of how an implementation can be done and the issues that need to be solved.

To implement the WCET estimation method, we have extended an existing instruction-level simulator, PSIM [Cag], with the capability of handling unknown values and by adding the WCET algorithm described in Section 2.4.2 to control the path exploration and merging. An overview of the system can be seen in Figure 4.1 and in the rest of this chapter we will describe in more detail the implementation of the different parts of the tool. The next section focuses on the WCET algorithm. Then, in Sections 4.2 and 4.3, we explain how the PSIM simulator has been extended and how some issues regarding the implementation of merging work. Finally, in Section 4.4, we define the concept of path progress before we end the chapter in Section 4.5 by discussing the strong and weak aspects of the tool.

4.1 WCET algorithm

The WCET algorithm is built around the central data type holding the list of uncompleted paths. Each of these paths represents a simulation of a part of the program from the beginning of the program to some place in the middle of the program at a conditional branch whose outcome is unknown. To be able to continue the simulation of a path, a path contains a complete state of the system, i.e., the register and memory content, and the state of the timing model. Furthermore, each path contains information about the progress of the path, which is mainly a record of how many loop iterations that have been done.

To continue the simulation, the path that has made the least progress, *a minimum progress path*, is selected. The system state of this path is copied to the state of the PSIM simulator and then the simulator is told to run. The simulator continues simulating until an unknown conditional branch is found or the end of the program is reached. When an unknown conditional branch is found, the simulator state is copied back to create two new paths, p_n and p_t. The unknown condition for the branch instruction in the paths p_n and p_t is set so that the branch will be non-taken and taken, respectively, whenever the simulation of the paths is continued. Finally, a new path is chosen for simulation. If two paths are found to have similar progress, *equal progress paths*, i.e., they have passed all loop headers an equal number of times and are at the same place in the program, these paths are merged before continuing. For performance reasons, the system state is never really copied to and from the simulator, but instead, pointers are copied. This makes the switch from simulating one path to another path quite fast.

To handle the update of the path progress information (see Section 4.4) and to support collecting statistics and doing experiments, we have added the possibility to register a callback-function on an arbitrary instruction address. Whenever the simulator fetches an instruction from such an address, the callback-function is called before continuing the simulation. This has proven to be a convenient way to interact with the simulator. For example, a trace dump to a file can be started when the simulator enters a specific region. Also, it is possible to flush or change the timing model at an arbitrary location, used, for example, to do timing analysis of single functions.

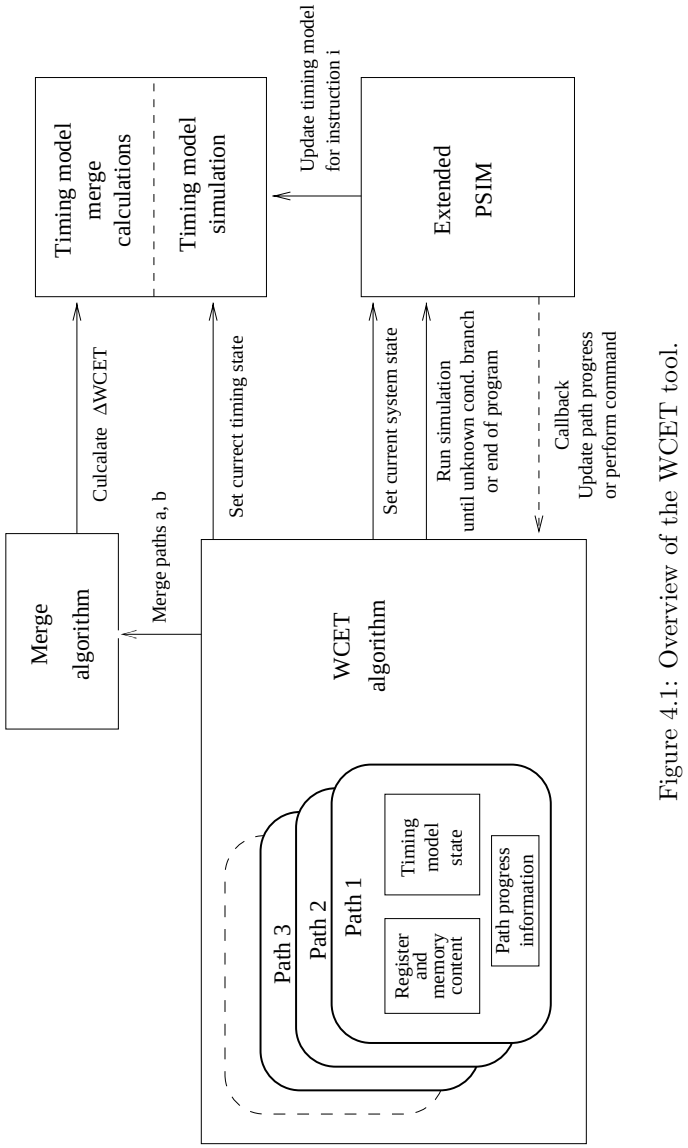


Figure 4.1: Overview of the WCET tool.

4.2 The PSIM simulator

The PSIM simulator is included in the distribution of GDB (The GNU Project Debugger) as a simulator target for executing and debugging PowerPC code. It was chosen by us due to its relative simplicity. The simulation is done in an interpreter loop that first fetches an instruction and then dispatches it to a function implementing the semantics of this instruction. Most of the semantic functions use fairly simple arithmetic operations involving the processor registers.

To extend the simulator, which is written in the C language, we made minor modifications to the simulator code to make it work in a C++ environment. This was made in order to exploit the fact that arithmetic operators can be overloaded in C++. We changed the data type for registers and memory to a new extended version of the same type. This extended data type (a C++ class) added the possibility of expressing the value *unknown* and also defined all arithmetic operators on this new type to correctly perform calculations involving *unknown*. As a result of this, the semantic functions could be reused with minor modifications. This greatly simplified the implementation and also prevented new bugs to appear in the semantic functions. To further simplify the implementation, we chose not to support floating-point operations with unknown values. The semantic functions for the floating-point operations are quite complex but we believe the same approach of redefining the register data type could be used to extend floating point functions as well.

The definition for the extended data types for registers and memory differ. For the memory, one additional bit of memory is used for each 32-bit word to hold the known/unknown information. Also, one additional bit is used to hold lock/unlock information necessary to handle unknown load and store instructions. All memory words belonging to an unpredictable data structure are marked as locked before starting simulation. When a memory word is locked, writes to this word are ignored and reads always return *unknown*. For registers we used one additional bit of storage for every bit in the register. Thus, each bit in a register can be set to known or unknown. This finer granularity is useful especially for the condition code register which contains different condition codes in different bit fields. Bit granularity for memory was not chosen in order to save space.

Other important modifications to the original simulator code are the

possibility of switching the register and memory state by simply switching a pointer, adding a test to the semantics of conditional branches to check if the branch condition is unknown, in which case the simulation should stop, adding a test to the semantic function for store instructions to notify the user of all store instructions having an unknown reference address, and finally adding a call in the interpreter loop to call the update function for the timing model.

4.3 Merging

The merging of two paths p_A and p_B is done by first doing a merge of the timing states, t_A, t_B, and then merging the content of registers and memory. The path resulting from the merge replaces the path that was found to be the long path and the short path is deleted.

To speed up the merging of paths and the creation of new paths, we only copy memory content when needed. We use the fact that paths that are to be merged, often have shared a long history of execution before they got split up. By only recording changes made to the system state since the time where the two paths were created, it is possible to quickly identify the parts of memory where the two system states differ. This is implemented by dividing the 1 Mbyte main memory into small fixed sized chunks of 512 bytes and letting each path store only modified memory chunks. In this way, only a few chunks of memory need to be compared during a merge operation and when creating new paths, no memory needs to be copied.

When only storing modified memory chunks we must also take into account that paths can split several times. This makes it necessary to represent the memory as a tree where each node holds modified memory chunks compared to its parent node. The leaf nodes of this tree represent the actual memory content of the current paths. An illustration of this is shown in Figure 4.2. Whenever a write occurs to a memory chunk that is not present in a leaf node, this memory chunk is copied to the leaf node from a parent node.

The tree structure used to hold the memory content has proven to be useful as a general structure to hold information that is changed in an incremental way during simulation, since such information can partly be shared between paths. For example, the cache conflict analysis that

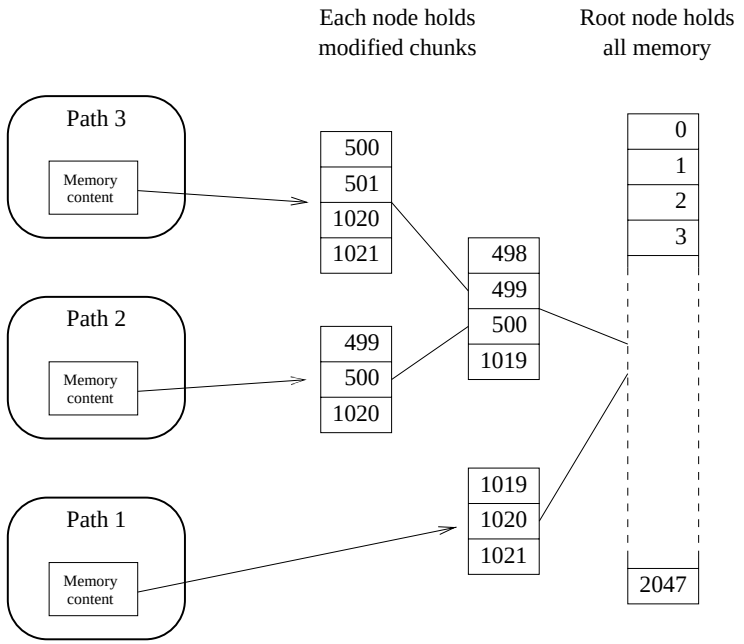


Figure 4.2: The memory content is represented as a tree where each node holds modified memory chunks compared to the parent node.

will be presented in Chapter 8 needs a list of memory accesses for each path. This list is also stored using the same tree structure as the memory content.

4.3.1 Cache algorithm

Before leaving the subject of merging we will present the algorithm used for calculating the upper bound Δ_{IC} (and Δ_{DC}) as presented in Section 3.3. The algorithm is based on the alternative expression from Section 3.3:

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

where $c(s,i)$ is defined by:

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

Expressed informally, this means that we can get an extra miss from IC_A when compared to IC_B if a block $IC_B(s,i)$ is not found in IC_A or when this block can be replaced earlier in IC_A than in IC_B. The algorithm calculates the number of extra misses, CA for one cache set s. Based on this, the upper bound Δ_{IC} is obtained by:

$$\Delta_{IC}(IC_A, IC_B) = P_{IC} \sum_{s=0}^{n-1} CA(IC_A(s), IC_B(s))$$

The algorithm, which is presented in Figure 4.3, uses the boolean vector *found* to keep track of the cases where blocks found in both IC_A and IC_B can be replaced earlier in IC_A than in IC_B. This vector is gradually updated in the loop going through the tags in IC_B. For example, consider a vector *found* containing the following in some iteration of the loop:


```
function CA(IC_A(s), IC_B(s))
    misses = 0
    found[0...(m-1)] = false
    first_false = 0
    for all tags b ∈ IC_B(s)
        if b = invalid then
            return misses
        else
            search for tag b in IC_A(s, [first_false, ..., (m-1)])
            if tag b found at position k then
                found[k] = true
                if k ≠ first_false then
                    misses = misses + 1
                end if
            else
                misses = misses + 1
            end if
            advance first_false if needed to point at
                first false element in found
        end if
    end for
    return misses
end function
```

Figure 4.3: The algorithm used to calculate Δ_{IC} (and Δ_{DC}). It marks in a vector *found* the position of tags in IC_A processed so far. A newly processed tag in IC_B must be found at the first false position in this vector to not count as an extra miss.

i	0	1	2	3	4	5	6	7
$found[i]$	true	true	true	false	false	false	false	false

This means that the three first tags in IC_A are all found in IC_B below or at the position in IC_B that is currently being processed. Let us now assume that the next tag considered from IC_B (at position i in $IC_B(s)$) is found at position 4 in IC_A. Then, $found[4]$ would be set to true. However, $found[3]$ = false. This means that $IC_A(s, 3)$ contains a tag not yet found in $IC_B(s)$. Thus, there exists a block $g < 4$ such that: $IC_A(s, g) \neq IC_B(s, h)$ for all $h < i$ which means that this block can be replaced earlier in IC_A than in IC_B. In order for a block to not count as an extra miss, the tag b from IC_B must be found in IC_B at the first false position according to the vector *found*.

4.4 Path progress

An important concept for the viability of the WCET algorithm is the concept of path progress. The WCET algorithm needs path progress information to determine when paths are possible to merge and which path to simulate next. Two paths that are allowed to merge are defined as being *equal progress paths*. All paths that have made the least progress and should be simulated further are defined as *minimum progress paths*. The current implementation of these concepts define path progress in terms of the number of loop iterations done so far and need information about the location of loops in the program. Ideally, this information should be extracted automatically from the control flow graph of the program, but to simplify the implementation and to avoid the need of constructing a control flow graph, manual annotations are used instead to mark loop headers and loop exit paths, and to give reachability information. These annotations can be added on a per-need basis. Using no annotations works fine if the analyzed program contains a single feasible path. Then, no path merging will be needed. On the other hand, if the number of simulated paths grows exponentially, we need to add annotations for all loops that surround the unknown conditional branch causing the exponential growth.

The currently used progress definitions are based on the assumption that all loops in the program can be related to each other in the form of a loop hierarchy tree. An exampel of such a tree can be seen in Figure 4.4.

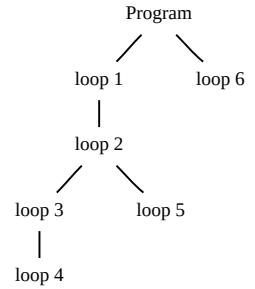


Figure 4.4: Example of a loop hierarchy tree which shows how loops are nested within each other.

This tree shows whether two loops are disjoint or one loop is nested within the other (the parent). To construct such a tree, we imagine that we have the complete control flow graph of the program. Moreover, functions are inlined at each calling location to distinguish between different instances of the function calls. Recursive function calls can be treated as equivalent to loops. To form a hierarchy tree of the loops present in this expanded control flow graph, we assume that the graph only contains natural loops [ASU86] and if two loops share the same loop header, they should be treated as one loop. For programs containing more complex loop structures, the WCET algorithm still works but the merging of paths is maybe done less often, which could lead to an increased number of paths to analyze.

4.4.1 Progress representation

In each path, the progress information is represented by a stack of loop counters. This stack holds information about how many times the current loop and its direct and all indirect parent loops have been entered. Thus, the elements in a stack corresponds to all loops along a path from the root to the current loop in the loop hierarchy tree. Each item in the loop stack, i, holds the id of the loop, $i.id$, and a count, $i.cnt$. For example, consider the stack (the rightmost element being at the top):

$$A = (a_1, a_2, a_3) = \left(\begin{array}{ccc} id\ 1 & , & id\ 2 & , & id\ 5 \\ 3 & & 1 & & 6 \end{array} \right)$$

This stack shows a path which has entered *loop 1* three times, *loop 2* one time and *loop 5* six times. The last instruction simulated is inside *loop 5*. To keep the loop stack updated, the simulator must push a new counter on the stack when a new loop is entered, and pop off the counter when the corresponding loop is left. If a loop is reached that is already in the stack, its counter must be incremented. For this to work, the simulator relies on the manual annotations to know the location of all loop headers and the loop exit paths.

An annotation marks the location of a loop header or the location of an exit path (actually the location of the destination of the exit path) and also gives an identification number to the loop, id, and the identification number of the parent loop, id_p, (the loop surrounding this loop). Top-level loops have an: $id_p = 0$. At the start of the simulation, the annotations are parsed and each annotation is connected to a specific instruction address using compiler debug information. At each of these locations, the simulator is asked to register a *progress update callback*, i.e., whenever the simulation passes one of these locations, a function is called that updates the progress information of the current path.

Whenever a loop header is passed, the progress update callback updates the progress based on the loop identification, id, associated with the current instruction address and based on the loop counter on the top of the stack, t, as follows:

if $id = t.id$ **then**
  same loop: increase $t.cnt$
else
  if id is a child loop of $t.id$ **then**
    push new loop counter on stack
  else
    pop top loop counter, try again with new top item t
  end if
end if

If the stack is empty, a new loop counter is always pushed onto the stack. Newly pushed counters start counting at 1. If a loop is left then the counter corresponding to this loop id and all counters above this counter in the stack is popped off the stack. This makes it possible to leave several loops using only one annotation. Also, the loop exit annotations

are seldom needed. If a new loop is entered before any unknown conditional branch has stopped the simulation, the old loop's counter will automatically be popped off according to the loop header update function described above.

4.4.2 Path progress definitions

Based on the loop hierarchy tree and the loop counter stack it is now possible to formally define the concept of path progress. We will assume that the loop counter stacks are updated to record the exact progress made, although in practice, annotations can often be left out without degrading the analysis.

We will begin by defining a relation between stacks. To ease the presentation, we introduce the following relations between two stack elements a_i and b_i:

$$\begin{aligned} a_i = b_i &\Leftrightarrow a_i.id = b_i.id \wedge a_i.cnt = b_i.cnt \\ a_i < b_i &\Leftrightarrow a_i.id = b_i.id \wedge a_i.cnt < b_i.cnt \end{aligned}$$

Now, consider two stacks $A = (a_1, a_2, \ldots, a_p)$ and $B = (b_1, b_2, \ldots, b_q)$ belonging to paths p_A and p_B, respectively, where p and q are the number of elements in each stack. To define relations between these stacks, we first set m to the number of top-most loops that A and B have in common. This means that the paths have reached some locations that are inside the loop $a_m.id$ (which is the same as $b_m.id$) but can be at different locations inside this loop. Now, we can define:

$$\begin{aligned} A = B &\Leftrightarrow m = 0 \vee a_i = b_i, \text{ where } 1 \leq i \leq m \\ A < B &\Leftrightarrow (a_1, \ldots, a_m) < (b_1, \ldots, b_m) \end{aligned}$$

where:

$$\begin{aligned} (a_1, \ldots, a_i) < (b_1, \ldots, b_i) \Leftrightarrow\ & i > 0 \wedge (a_1 < b_1 \vee \\ & (a_1 = b_1 \wedge (a_2, \ldots, a_i) < (b_2, \ldots, b_i))) \end{aligned}$$

The meaning of these relations is as follows. If $A < B$ then we now for sure that path p_A has made less progress than p_B since p_A has made fewer iterations in some loop compared to p_B. If $A = B$ then both paths p_A and p_B have made the same number of iterations in all top-most loops they have in common, but path p_A may have entered some nested loops that p_B has not entered or vice versa. Both paths are somewhere inside the body of the loop $a_m.id$. This body corresponds to a subgraph of the control flow graph and we will call this subgraph G. To really pinpoint the progress for the case when $A = B$, we must also involve the program instruction counters or using PowerPC terminology, the current instruction addresses, cia_A and cia_B, in the system states of p_A and p_B, respectively. These addresses correspond to some nodes in the graph G. We assume that cia_A and cia_B corresponds to node n_A and n_B in the graph G, respectively. We then define the following relation:

$$n_A < n_B \Leftrightarrow n_A \text{ can reach } n_B \text{ in graph } G$$

Finally, we can extend the relations between stacks to form the following relations between the paths p_A and p_B:

$$\begin{aligned} p_A = p_B &\Leftrightarrow A = B \wedge cia_A = cia_B \\ p_A < p_B &\Leftrightarrow A < B \vee (A = B \wedge n_A < n_B) \end{aligned}$$

The relation $p_A < p_B$ defines a partial ordering of paths. A path p_A is *a minimum progress path* iff there exists no other path, p_B, so that $p_B < p_A$. Also, two paths p_A and p_B are *equal progress paths* iff $p_A = p_B$.

The implementation does not follow these definitions fully. Instead of using a control flow graph (which we do not have available) we simply define:

$$n_A < n_B \Leftrightarrow cia_A < cia_B$$

That is, we assume that a location in the program can be reached from all other locations that have a lower instruction address. Thus, we only get an approximation of the minimum progress path definition above, which could reduce the number of merge opportunities. To cover possible problems with this approximate implementation, we also include the possibility to add reachability information by means of manual annotations.

For all programs used in the experimental evaluation in the next chapter, this approximate definition proved to be sufficient. However, some reachability annotations were needed.

4.5 Discussion

The implementation of the tool has proven to us that the design can be quite modular. For example, the WCET algorithm implementation would not change in any major way if another simulator (for example, simulating another architecture) was adopted. Also, the timing model can be changed without causing any changes in the WCET algorithm or simulator code. Adding timing analysis for new hardware mechanisms is also quite straight forward. The timing simulation model must be extended with new timing state and new update methods, and the merge calculation must be extended with a new method calculating the $\Delta WCET$ contribution for the new mechanism.

In Chapter 2, we discussed the advantages of a more powerful value domain. There is no inherent problem in extending the simulator to work with a more powerful domain. The result would be a slower simulation that needs more memory. Our choice of domain results in an additional one bit of memory for each 32-bit word of memory to hold the known/unknown status. An interval representation, for example, would need two extra words for each word of memory. Also, a more complex semantics would be needed, which would result in a slower execution of each instruction. On the other hand, the more complex domain might be preferable for some applications, if it manages to cut more infeasible paths and thereby gain speed and accuracy compared to our simple domain.

The greatest need for improvement is in the handling of path progress. The use of manual annotations is cumbersome and a more automated method would be preferable. Ideally, the loop information needed for the path progress relations could be extracted from the binary code as a byproduct of the simulation. However, it is maybe not enough to solely base such a method on the unknown conditional branches. It is possible that we need to create the control flow graph and find out how loops relate to each other and must therefore instrument all branch instructions. This is a quite complex matter which is outside the scope of this thesis. Furthermore, the progress measure adopted here is mainly suitable for C like

languages. For more dynamic languages or other run-time environments, other progress measures must probobly be developed.

Another nice improvement would be an integration with the GDB (the GNU Project Debugger). This would permit a mixed debugging/analysis framework where one could use GDB to insert unknown values at specific points, do an analysis of a small part of a program, and afterwards, study the result. However, issues regarding the user interface has not been further studied.

Chapter 5

Experimental Results

In order to evaluate the analysis accuracy of the cycle-level symbolic execution method, we have estimated the WCET of seven programs run on two different processor architectures making it possible to evaluate the path and timing analysis capability in isolation. While both architectures support the same instruction set—a subset of the PowerPC instruction set [IBM]—the first one assumes that all instructions execute in a single cycle with a zero-cycle memory access time. In contrast, to focus on the accuracy of the timing analysis, the timing model of the second architecture corresponds to the system presented in Section 3.2.

5.1 Benchmark programs and metrics

An overview of the seven programs can be seen in Table 5.1. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *DES*, *jfdctint*, and *compress*.[1] The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the programs. No optimization was enabled. The simulated run-time environment contains no operating system; consequently, we disabled all calls to system functions such as I/O in the programs.

In order to make a useful comparison how good estimates our method

[1]The programs *fib*, *isort*, and *jfdctint* is from the SNU-RT Benchmark Suite created by Sung-Soo Lim, Seoul National University. The programs *matmult*, *bsort*, and *DES* was provided by David Whalley, Florida State University. The program *compress* was taken from the SPEC CPU95 benchmark suite.

Name	Description
matmult	Multiplies two 50x50 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
DES	Encrypts 64-bit data
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

Table 5.1: Characteristics of the programs used.

produces, we need to calculate the actual WCET of each program. The actual WCET was determined by running the programs on the simulator with the worst-case input data. This was straightforward to determine for all programs except *compress*, where the worst-case input data was hard to find. Instead, a random sequence of 50 bytes were used as input.

Another evaluation issue is that two of the programs, *fib* and *compress* actually have a termination condition that depends on input data. Therefore, we needed to bound the number of iterations to make WCET statically decidable. Here, we used the approach described in Section 2.5 and added an extra exit condition in the loops. In *fib* we added the condition: $i \leq 30$ because we know that input data is always in this range. In *compress* we bound an inner loop whose iteration variable is j, using the current iteration count, i, of the outer loop: $j \leq i$. This is a safe but pessimistic bound, but we found it difficult to prove that a tighter bound could be used. The inner loop implements a secondary probe in a hash table and the number of iterations depends on unknown input data in a complex manner.

5.2 Path analysis results

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

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

Table 5.2: The estimated WCET using the ideal architecture.

In Table 5.2, we show three WCET numbers for each program: actual, estimated, and structural. The actual WCET is our measurement of the actual WCET as explained in Section 5.1. The estimated WCET is the WCET determined by the method and the structural WCET is the execution time of the longest structural path, including all infeasible paths, in the control flow graph of the program when using fixed bounds on the number of iterations of all loops. This number represents what a WCET method would estimate that does not eliminate any infeasible paths and uses fixed iteration bounds for all loops. The purpose of doing this is to analyze the capability of the method to eliminate infeasible paths. In the table, WCET is expressed in clock cycles and ratio is the estimated (or structural) WCET divided by the actual WCET.

For all benchmarks, except *compress*, we can see that the method succeeds in finding the actual WCET. In *compress*, the overestimation is caused by the inner loop. As mentioned previously, we bound this loop using the pessimistic condition $j \leq i$, but during a normal run, we have found that this inner loop is actually only doing one single iteration. It should be mentioned that for *compress*, we do not know if the WCET we determined is the actual WCET. We suspect that the actual WCET we use is lower than the real one.

Two of the programs, *matmult* and *jfdctint*, have no infeasible paths at all, and only one path is simulated. In *DES*, however, there exist infeasible paths caused by data dependencies between different functions. These infeasible paths are eliminated by our method and only one path is simulated. In *bsort* and *isort*, all infeasible paths were not eliminated.

Still, this does not lead to any overestimation, since all simulated infeasible paths are shorter than the worst-case path the method finds.

If we take a look at the estimated structural WCET of the programs, we see that the WCET is grossly overestimated for *bsort*, *isort*, and *compress*. In *bsort* and *isort* it depends entirely on using a fixed iteration count for an inner loop which is normally bound by the current iteration count of the outer loop. This leads to an overestimation of a factor of two of the execution time for the loop and affects *bsort* more than *isort* because of the greater number of iterations done in *bsort*. In *compress*, there is a similar inner loop which is forced to have a fixed iteration bound, again causing an overestimation of a factor of two. In addition, there exists a very long infeasible path that extends the structural estimate. As for *DES*, the tiny overestimation results from infeasible paths. As shown in Table 5.2, our method successfully manages to extract the loop bounds and eliminates the infeasible paths automatically.

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

5.3 Timing analysis results

In this section, we analyze how well our integrated path and timing analysis method manages to estimate the WCET of each program. We now assume the detailed timing model of the PowerPC architecture described in Section 3.2 with timing parameters according to Figure 5.1. Table 5.3 shows three WCET numbers: actual, estimated, and conservative. Again, actual is the actual WCET we have determined whereas estimated WCET is the WCET determined by our method. Finally, conservative WCET corresponds to the estimated WCET when caches are turned off and each instruction proceeds through the pipeline one at a time. Finally, the two ratios shown correspond to estimated and conservative WCETs divided by the actual WCET, respectively.

Starting with all applications except *DES* and *compress* we note that

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

Table 5.3: The estimated WCET with caching and pipelining enabled. The ratios for the cache-all versions of *DES* and *compress* include the slowdown resulting from not caching accesses to unpredictable data structures.

Instruction and Data Cache
256 byte size
direct−mapped
16 byte/block
10 cycles miss penalty

Pipeline Latencies	
LSU	2 cycles
IU	1 cycle
MCIU	4 cycles

Figure 5.1: The parameters used for the timing model when evaluating the timing analysis. LSU is the load/store unit, IU is the integer unit, and MCIU is the multiple-cycle integer unit.

our method manages to make an **exact** estimate of the actual WCET. This is somewhat unexpected even if these same programs were perfectly analyzed with respect to path properties. The expected added complication now stems from two sources: merging of the timing state and the use of a data cache. We will now take a closer look on these sources in order to understand the reason for the good result.

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

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

DES and *compress* contain unpredictable data structures as defined in

Section 2.3. In our method, we avoid the pessimism of charging two misses by using an alternative approach. Since unpredictable data structures are mapped into a special area of the memory, as discussed in Section 2.3, we also mark this area as non-cacheable. This is supported by most processors, since memory mapped I/O locations are in general not cached. During analysis we then know that an unknown reference will at most cause a single cache miss. Further discussion about unpredictable data accesses can be found in Chapter 7. For comparison purposes we have included two versions of *DES* and *compress*: one where only accesses going to predictable data structures are cached — called cache-predictable version — and one where all accesses are cached — called cache-all version. WCET estimation has only been done for the cache-predictable versions. These numbers are shown in Table 5.3. The ratios shown for the cache-all versions of *DES* and *compress* are the estimated and conservative WCET for the cache-predictable versions divided by the WCET for the cache-all versions. It should also be mentioned that when data caching is enabled it is hard to find the worst case input data for the cache-all versions of *DES* and *compress*; the data addresses going to the data cache sometimes depend on unknown input data. We have not made any effort to address this problem.

As can be seen from Table 5.3, an almost perfect estimate of the WCET of *DES* is determined by our method in spite of unpredictable data structures. The numbers for the cache-all versions of *compress* and *DES* reveal to what extent the unpredictable data structures affect the estimation. In *DES*, only 0.6 % of all data accesses are directed to unpredictable data structures resulting in a slight overestimation. On the other hand, in *compress* 34 % of all accesses are directed to unpredictable data structures. By not caching these accesses we increase the execution time (and the overestimation) by 18 %.

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

To fully realize the importance of doing timing analysis, we can take a look at the conservative WCET in Table 5.3. We see that when treating all cache accesses as misses and permitting no pipelined execution we

	Analysis time [sec]		Number of	Number of
Program	Ideal	Detailed	merges	instructions
matmult	8.7	16.3	0	7063912
bsort	2.3	3.0	4949	292026
isort	< 0.1	< 0.1	36	2594
fib	< 0.1	< 0.1	0	697
DES	0.2	0.3	0	118675
jfdctint	< 0.1	< 0.1	0	6010
compress	2.1	2.4	3965	49046

Table 5.4: The WCET analysis times for the ideal and the detailed timing model. Also included are the number of merges performed and the number of instructions simulated in the worst-case path found.

increase the overestimation of the WCET by a factor of between 5.6 and 10.6. In *compress*, for example, we find that the additional overestimation when doing no timing analysis is a factor of 6.1. Additionally, as we saw in the previous section, the overestimation when doing no path analysis (the structural WCET) is a factor of 3.3. Together, we get a factor of 20 in additional overestimation for *compress* when ignoring both path and timing analysis which clearly shows the importance of integrating accurate path and timing analysis.

5.4 Time complexity

Table 5.4 shows the time taken to perform the analyses of all benchmark programs, as measured on a SunBlade 1000 workstation (UltraSPARC-III, 600 MHz). The simulator simulates approximately $7/16.3 = 0.4$ million instructions/second when using the detailed timing model. For the programs where no merging occurs, the simulation speed directly determines the analysis time. However, when merging occurs, many paths besides the worst-case path is being simulated and time is also spent doing the merge operation. This is the reason for the relatively long analysis times for *bsort* and *compress*.

5.5 Discussion

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

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

Chapter 6

Unbounded Timing Effects

In the previous chapters, we have seen how to do WCET analysis for a timing model containing an instruction cache, a data cache, and a simplified pipeline using in-order dispatch. An interesting question is what would happen if we try to extend the timing model to model a pipeline that supports dynamic scheduling. In this chapter, we will mainly focus on the problems of analyzing dynamic scheduling of instructions but also study the problems of analyzing other mechanisms.

As far as the simulation of the pipeline is concerned, the timing model used can be arbitrarily complex. Limitations of the timing model is introduced by the requirements of the merge operation—it must be possible to calculate a constant upper bound, $\Delta WCET$, quantifying the difference between two timing states. An upper bound, $\Delta WCET(t_A, t_B)$, is a function that compares the timing state, t_A, in one path with the timing state, t_B, in another path. This upper bound is calculated by comparing two timing states. However, it can also be interpreted in terms of changes to the timing state of a system. If the timing state, t_1, is changed into a new state, t_2, then $\Delta WCET(t_2, t_1)$ is an upper bound on the effect of these changes on the future execution time. We can use this to classify different hardware mechanisms based on how easily they can be analyzed:

Definition 6.1 *An architectural mechanism influencing the timing of instructions is said to have* **a bounded timing effect** *if there exists an upper bound $\Delta WCET(t_2, t_1)$ on every possible change of the timing state concerning the mechanism from t_1 into t_2. Conversely, a mechanism has* **an unbounded timing effect** *if there exists no such (constant) upper bound.*

We will also use the definitions *bounded mechanism* and *unbounded mechanism* as shorthand for mechanisms having a bounded or unbounded timing effect, respectively. A system where all mechanisms have a bounded timing effect can be efficiently handled by our method, but if one or several mechanisms have an unbounded timing effect our merge operation can no longer be used to guarantee the safety of the final estimated WCET. Moreover, even if there exists an upper bound on the timing effect, we must also be able to calculate it.

A common assumption is that if the worst-case instruction execution time is assumed, the WCET estimation will be safe. We define a *timing anomaly* as a situation when such assumptions do not hold. In the next section, we first identify the possibility of *timing anomalies*, in dynamically scheduled microprocessors and show several examples of such anomalies. Because of these anomalies, dynamic scheduling is a mechanism that can have an unbounded timing effect. We will give other examples of mechanisms that have an unbounded timing effect in Section 6.2. In Section 6.3, we discuss the problem of analyzing unbounded mechanisms and explain why all previously published timing analysis methods for cache and pipeline analysis [HWH95, WMH+97, LMW95, LMW96, OS97, LBJ+94, KMH96, FHL+01, FW99, WE00] would result in prohibitive computational complexity to analyze these mechanisms. We will then, in Section 6.4, discuss possible solutions and present two examples of methods that can handle unbounded mechanisms, the program modification method and the serial execution method. We evaluate these methods experimentally in Section 6.5 before we conclude this chapter in Section 6.6 with a discussion about cache and pipeline related task scheduling issues.

6.1 Timing anomalies in processors

In this section, we will study the use of dynamic, out-of-order scheduling of instructions. The dynamic scheduling is complex to analyze since the scheduling of future instructions in the pipeline is dependent on the execution time of each individual instruction which can take one of many discrete values depending on input data. One example is a load instruction whose execution time depends on whether the address hits or misses in

the cache. Another example is an arithmetic instruction whose execution time may depend on the operands. Thus, the scheduling of instructions is typically dependent on unknown input data.

6.1.1 Definitions

Consider the execution of a sequence of instructions. For clarity reasons, we will use the term *latency* meaning the instruction execution time. When we use the term *execution time* it will mean the overall execution time of the program. Let us study two different cases where the latency of the first instruction is modified. In the first case, the latency is increased by i clock cycles. In the second case, the latency is decreased by d cycles. Let C be the future change in execution time resulting from the increase or decrease of the latency. Then:

Definition 6.2 *A timing anomaly is a situation where, in the first case, $C > i$ or $C < 0$, or in the second case, $C < -d$ or $C > 0$.*

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

The definition above allows for a broad range of effects. For example, Schneider and Ferdinand [SF99] have identified an acceleration effect, i.e., the penalty from an instruction cache miss can get extended when instructions are grouped for multiple issue. This acceleration effect is an example of a timing anomaly. However, in this thesis, we will only focus on timing anomalies related to the dynamic scheduling of instructions[1].

To model the instruction execution in a pipelined processor, one often uses a resource model. In this model, whenever an instruction that proceeds through a pipeline gets stalled, it is due to resource contention with another instruction that accesses a common resource or operand. Typical examples of resources are functional units and registers, but also buses, read and write ports, and buffers should be treated as resources if they can cause instructions to stall. The term *dynamically scheduled processors* is often used to describe a processor for which instructions execute out-of-program-order. However, it is not the out-of-order execution that

[1]The previously published report of these anomalies [LS99c] was also focused on timing anomalies related to dynamic scheduling. However, as pointed out by Jörn Schneider [Sch99], this was not really clearly stated.

is the central issue here. Rather, it is the order in which resources are allocated in the processor.

The resources that an instruction can use can be divided into *in-order* and *out-of-order resources*. In-order resources can only be allocated in program order to instructions. Out-of-order resources can be allocated to instructions dynamically, i.e., a new instruction can use a resource before an older instruction uses it according to some dynamic scheduling decision. Typical out-of-order resources are functional units that service instructions dynamically (out-of-order initiation). Examples of in-order resources are such registers that must be reserved in-order to guarantee that data dependencies in the program are not violated. Given this definition, it is now possible to state a condition when a processor can suffer from anomalies:

Statement 1 *If a processor contains out-of-order resources, timing anomalies related to dynamic scheduling of instructions can occur.*

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

6.1.2 Timing anomaly examples

The focus of our study will be the model of an architecture seen in Figure 6.1 based on a *simplified* PowerPC architecture containing no floating point units. A more realistic model is expected to contain more features that would result in out-of-order resource allocation. Even for this simplified architecture, timing anomalies do show up.

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

All resources in the modeled processor are considered to be in-order resources except the integer unit (IU) and the multiple-cycle integer unit (MCIU) which are out-of-order resources. The load/store unit (LSU)

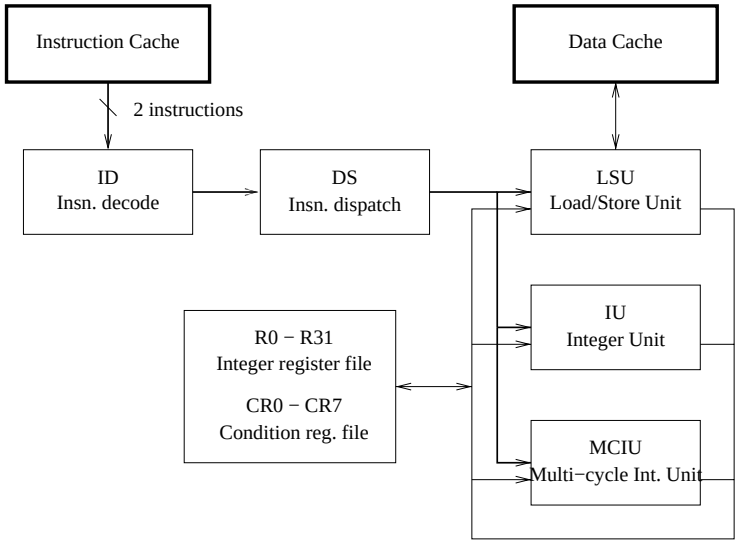
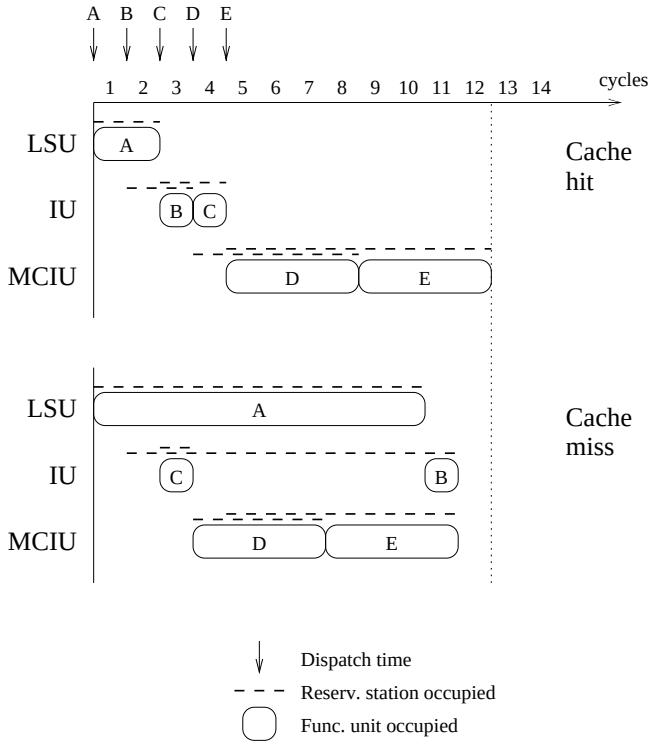


Figure 6.1: A simplified, yet timing-anomalous, PowerPC architecture.

often initiates execution in-order to preserve ordering of memory accesses so we also treat it as an in-order resource here. The out-of-order resources, IU and MCIU, make timing anomalies possible as we will demonstrate in three examples: one showing that a cache hit may be worse than a cache miss, another showing that the miss penalty can be greater than expected, and a third showing a possible domino effect when executing loops.

Anomaly 1: Cache hits can result in worst-case timing

The first example presents a case where a data cache hit causes an overall longer execution time than a data cache miss. Consider the table in Figure 6.2, which shows a sequence of instructions (A-E) and in which clock cycle they are dispatched. The instructions represent the use of different functional units: the `LD rd,0(ra)` instruction uses the LSU, the `ADD rd,ra,rb` uses the IU, and the `MUL rd,ra,rb` uses the MCIU. Register `rd` is the destination register and `ra` and `rb` are the source registers. The registers create data dependencies and thereby an ordering between



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

Figure 6.2: Anomaly 1. An example when a cache hit causes a longer execution time than a cache miss.

instructions. To simplify the discussion of the examples we focus only on the functional units and their reservation stations. We assume that the instructions are dispatched according to the relative times seen in the instruction table in Figure 6.2 although in reality, on a dual-issue pipeline, additional instructions would be needed to make the instructions dispatch according to the example.

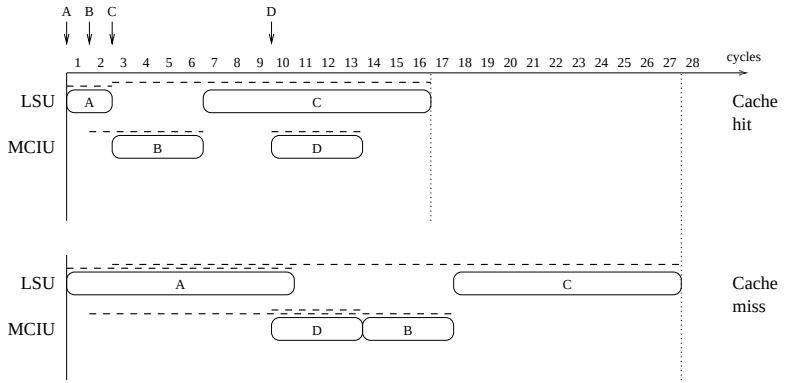
The diagram in Figure 6.2 shows when each functional unit is busy executing an instruction. Also shown as horizontal dashed lines is when the reservation stations are occupied. At the top, arrows indicate when each instruction is dispatched to the reservation stations. Two cases can be identified, one when the load address hits in the data cache and one when it misses the cache.

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

Anomaly 2: Miss penalties can be higher than expected

The second example shows that the overall penalty in execution time due to a cache miss can be higher than the normal cache miss penalty. Consider the instruction sequence in Figure 6.3. The first instruction is a load instruction which can either hit or miss in the cache. We assume that the second load instruction (C) always misses. The first three instructions: A, B, and C, depend on each other and must execute one at a time. In the cache hit case all instructions will execute as soon as possible. The last instruction, D, will not interfere with the execution of the other instructions.

If the first load experiences a cache miss, the execution of B will be postponed. In this unfortunate case, instruction D has already started when B becomes eligible for execution and B will be further postponed.



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

Figure 6.3: Anomaly 2. An example when the cache miss penalty is higher than expected.

The result of this is that instruction C will finish executing 11 clock cycles later in the cache miss case as compared with the cache hit case. This is greater than the normal cache miss penalty of 8 clock cycles. In this case, the anomaly is due to the MCIU being an out-of-order resource, which allows instruction B and D to execute in arbitrary order.

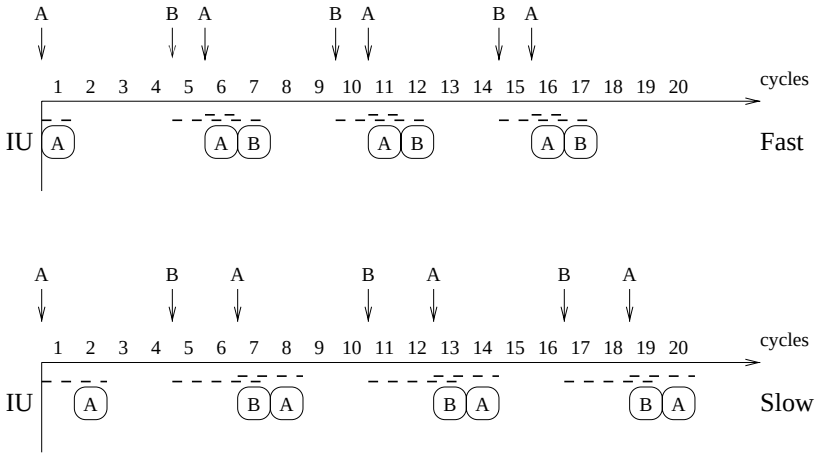
Anomaly 3: Impact on WCET may not be bounded

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

Consider the instruction sequence in Figure 6.4. The two instructions A and B constitute the body of a loop doing a number of iterations. The delicate execution scenario shown here demands special requirements on the dispatch and execute cycles. Therefore, the table entry for the dispatch clock cycle and the additional table entry for the execute clock cycle show the dispatch and execute clock cycle relative to a previous instruction. By E_A we mean the clock cycle when A executed in the previous iteration of the loop. By D_A we mean the clock cycle when A was dispatched in the current loop iteration.

The two different scenarios shown in Figure 6.4 are the result of dispatching and executing the two instructions A and B repeatedly according to the dispatch and execute cycle rules starting from two different executions of the first A instruction. In the fast case, instruction A in the first iteration executes immediately when it is dispatched. In the slow case, we imagine that it gets delayed one clock cycle because of a dependency with an earlier instruction. This delay in the beginning is enough to cause a domino effect that will delay the execution of A by one clock cycle in each iteration. The total penalty on the execution time, caused by the small delay of A in the beginning, will be k clock cycles if the loop does k iterations. In the slow case, we assume that the old B instruction gets priority over the new A instruction in each iteration.

In summary, we have shown three examples when timing anomalies may show up in dynamically scheduled processors. These anomalies were



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

Figure 6.4: Anomaly 3. Example of a domino effect.

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

6.2 Unbounded timing effects

The anomalies shown in the previous section can lead to unbounded timing effects. An example of this is the domino effect (anomaly 3). The number of extra cycles incurred is proportional to the number of loop iterations. Therefore, a constant upper bound on this effect does not exist. Moreover, even if we can prove that domino effects cannot arise, it is not clear how to calculate an upper bound on the effects of an anomaly.

Still, for more simple cases of dynamic scheduling, it should be possible to create upper bounds but this is still an unexplored area of research. For timing anomalies not related to dynamic scheduling, upper bounds seems to be more simple to construct. For example, the acceleration effect reported by Schneider and Ferdinand [SF99] can be handled by adding a constant number of cycles to the instruction cache penalty whenever the outcome of a cache access is unknown.

The unbounded effects we have found in a pipeline is due to scheduling anomalies. However, there are more simple examples of mechanisms having unbounded timing effects. Two examples are the use of *random* and *FIFO* (first in, first out) cache replacement strategies. Let us take a closer look at these mechanisms.

6.2.1 Random cache replacement

Using random (pseudo random) cache replacement sounds unpredictable and it is. Consider two cache timing states, IC_A and IC_B, belonging to two paths p_A and p_B, respectively. A cache state must now also contain a seed for the random number generator and we assume that IC_A and IC_B differ only in the seed. If we want to calculate $\Delta_{IC}(IC_A, IC_B)$, we get the worst case by considering a future access sequence that for all accesses will generate cache misses from IC_A. Such a sequence is easy to construct if we know how the random numbers are generated. Now, since the random number seeds differ in the two cache states, the same sequence of future accesses will not generate cache misses for all accesses from IC_B. Otherwise, it would be a very bad random number generator. Also, the longer we make the future access sequence, the greater the difference in cache misses we will get. A conjecture we make is that there exists no constant upper bound.

6.2.2 FIFO cache replacement

From a WCET point of view, the FIFO, first in, first out, replacement seems similar to LRU but it is not. An initial difference in timing state between two cache states can be sustained using FIFO whereas in LRU the timing state of two caches will eventually converge when updated with the same future access sequence. An example of this is shown in Figure 6.5 where a set s in two cache states IC_A and IC_B has an initial

	$IC_A(s,i)$		$IC_B(s,i)$	
i	0	1	0	1
Initial state	2	1	2	3
$a_0 = 3$	**3**	2	2	3
$a_1 = 1$	**1**	3	**1**	2
$a_2 = 2$	**2**	1	1	2
$a_3 = 3$	**3**	2	**3**	1
$a_4 = 1$	**1**	3	3	1
$a_5 = 2$	**2**	1	**2**	3
$a_6 = a_0$	...			
$a_7 = a_1$	...			
...				

Figure 6.5: Demonstration of unbounded timing effect for FIFO replacement. Two cache states are shown for one set s in a 2-way cache. For FIFO replacement, $i = 0$ is the most recently inserted block. Bold face block tags represent cache misses. The cache state IC_A causes misses for all accesses a_i and IC_B only for every second access.

state of: $IC_A(s) = [2\ 1]$ and $IC_B(s) = [2\ 3]$. Now, when updating these states with the access sequence $a_i = 3, 1, 2, 3, 1, 2, \ldots$, we get a number of extra misses from IC_A when compared to IC_B that is proportional to the length of the access sequence a_i. This shows that FIFO is a mechanism having an unbounded timing effect.

6.3 Limitations of previous methods

Unbounded mechanisms cannot be handled by our merge operation. In this section, we present arguments to show that all previously published timing analysis methods for cache and pipeline analysis will face severe problems when trying to analyze unbounded mechanisms. We will use dynamic scheduling with the possibility of a domino effect as an example of an unbounded mechanism and indentify the problems that arise if we want to perform accurate pipeline analysis. To correctly estimate the WCET, one would have to consider the effect all variations in instruction execution times have on the possible instruction schedules.

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

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

All previously presented methods for doing cache and pipeline analysis [HWH95, WMH+97, LMW95, LMW96, OS97, LBJ+94, KMH96, FHL+01, FW99, WE00] perform pipeline analysis by first looking at each instruction or basic block and then combining the WCET of all these entities into a total WCET for the whole program. While none of these methods are designed to handle dynamically scheduled processors, they nevertheless rely on the capability to make local safe decisions when regarding variable-latency instructions. For example, in [HWH95, TF98] the cache analysis is done first and then later used in a pipeline analysis step. Whenever it is not possible to classify a cache access as a hit or a miss, it is conservatively treated as a miss when doing the pipeline analysis. This may lead to a too optimistic estimation as we have seen in the first anomaly example according to Figure 6.2.

Consider next a program containing several feasible paths. The WCET is then the maximum WCET found among all paths and in order to find the WCET we would have to examine all paths in the program. This is, in general, not feasible and timing analysis methods again rely on the possibility of making local safe decisions to reduce the complexity.

When analyzing a small section of the program, e.g., a loop, the longest path in this section is chosen before doing the analysis of the rest of the program. Unfortunately, due to the anomalies, it is not possible to make local safe decisions. To see this, assume that the small section contains l different paths. When simulating the execution of the different paths in the pipeline we may end up with l different pipeline states, leading to the same problem as for the variable-latency instructions. It is not possible to know which pipeline state (path) that gives us the longest overall execution time.

An example of when local decisions are used to reduce the path complexity is the prune operation used in [LBJ+94, LHKM98]. It is used to discard some combinations of basic blocks that will execute in shorter time than another combination of basic blocks found. To make this pruning decision, one must know how the execution of some basic blocks will influence the execution of other parts in the program. Due to, e.g., the domino anomaly (Figure 6.4), this can be difficult or even impossible. The same problem exists in [HWH95] where the longest path is chosen in each iteration of a loop.

To conclude, when doing timing analysis in the presence of unbounded mechanisms, it is not possible to make safe local decisions, i.e., safe choices between the different pipeline states that an unknown event may give rise to. Fortunately, we will in the next section show two approaches that can make it possible for previously published timing analysis methods to handle unbounded mechanisms.

6.4 Handling unbounded mechanisms

In this section, we will present two approaches to estimate the WCET of a program running on a system containing unbounded mechanisms. Both approaches can be used together with previously published timing analysis methods. The first approach is to approximate the unbounded mechanism with a more pessimistic but bounded timing model. This approach can successfully be used as a way to handle both random and FIFO replacement by modeling the cache as a direct-mapped cache containing the same number of blocks as there are sets in the original cache. Recently, a technique was proposed by Ferdinand et al. [FHL+01] to model the Coldfire 5307 cache which uses a kind of global FIFO replacement. The

global FIFO replacement is also an example of an unbounded mechanism and they model it as a direct-mapped cache. Thus, they use a pessimistic timing model. For timing anomalies, we will present a serial-execution method to safely handle timing anomalies in dynamically scheduled processors.

The second approach to handle unbounded mechanisms is based on program modifications—by modifying the program we make it possible for timing analysis methods to rely on safe local decisions. We will also give an example of how this method safely can handle timing anomalies. At the end of this section, we present a case study of how the program-modification method can be used together with the cycle-level symbolic execution method. We only focus on instruction and data cache analysis and the out-of-order resource use of the functional units.

6.4.1 The pessimistic serial-execution method

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

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

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

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

It is probably possible to create a less pesimistic timing model that

still has a bounded timing effect. However, this has not been further studied in this thesis. It may be possible to relate the anomalies found in dynamically scheduled processors to the anomalies that have been found in the scheduling of tasks on a multiprocessor [Gra69, AJ02].

6.4.2 The program modification method

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

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

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

More generally, to handle any unbound mechanism we must arrange so that all unknown events and all path merges only result in one possible timing state. Ideally, we would like to have explicit program control of all internal state in a processor that may influence the future timing of instructions. However, for most processors, we have only a limited control of the timing state.

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

instruction completes. This instruction can be used as a way to force serialization together with a variable-latency instruction.

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

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

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

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

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

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

When preloading blocks, it is best to preload a block that will be needed somewhere along the worst-case path. Then, no unnecessary pessimism is added due to additional cache misses. In addition, it is often

best to place a preload instruction outside loops if possible to reduce the overhead. The best way to preload is a complex issue, which we do not investigate further in this paper. In the experimental evaluation, this information was derived manually (see Section 6.5).

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

6.4.3 Case study: symbolic execution method

We will now take a closer look at how the program modification method fits in the method presented in the previous chapters of this thesis.

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

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

6.5 Experimental evaluation

We have evaluated the amount of pessimism introduced when estimating the WCET of seven benchmark programs, using the two methods presented in Section 6.4: the pessimistic serial-execution method, and the program modification method. The modeled architecture is the one presented in Section 6.1, consisting of a dual-issue pipeline with instruction and data caches.

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

6.5.1 Methodology

The seven benchmark programs used are identical to the programs used in Chapter 5 (see Table 5.1). To estimate the WCET of the benchmark programs, the method described in Section 6.4.3 has been used. The timing model used in the WCET simulator is based on the model of the PowerPC architecture discussed in Section 6.1 with the timing parameters according to Figure 6.1. However, instead of a detailed simulation model of the pipeline, we use an analytical approach. During simulation, the functional unit latencies of the simulated instructions are added together with instruction and data cache miss penalties. This we call the serial time, T_{serial}. We then assume that the time T to execute the program on the dual-issue architecture is:

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

The relation between T and T_{serial} is obviously not this simple in reality. The above formula would represent the ideal situation of dispatching two instructions each cycle. This is often not possible in reality due to cache misses and pipeline stalls and is highly program dependent. Nevertheless, this formula makes it easy to compare the different estimation methods. When estimating the WCET our model automatically produces the pessimistic serial-execution estimate. The other estimates are derived by using the formula above.

When modifying the programs we used `sync` instructions to handle the pipeline state and preload instructions to handle the instruction and data cache states as described in Section 6.4.3. We assumed that a single `sync` placed at a merge point in the program incurs a penalty of 5 cycles in the dual-issue architecture. When one `sync` instruction is placed before and one after a variable-latency instruction, we assumed a penalty of 8 cycles, i.e., the second `sync` incurs less penalty than the first one since the pipeline is already flushed by the first `sync`. When adding preload instructions, the program becomes bigger. The effect of this on the latency and possible additional instruction cache misses has been estimated manually and accounted for in the results. Three integer multiply instructions were assumed to be variable-latency: `mulhw`, `mulhwu`, and `mullw`. The multiply immediate instruction, `mulli`, and all other instructions were assumed to have fixed latencies.

6.5.2 Evaluation results

The results from our evaluation of the seven benchmark programs can be seen in Table 6.1. The actual WCET has been determined by simulating the program using the worst-case input data, or using random input data if the worst-case input was to complex to determine. The table also shows the estimated WCET when using the serial method and when using the modified program method. Also included for comparison purposes is the unsafe program estimate, i.e., the dual-issue timing model has been assumed but no program modifications have been made. This is unsafe since timing anomalies can lead to an underestimation of the WCET. The ratio columns in the table is the estimated WCET values divided by the actual WCET. The modified slowdown is the modified program estimate divided by the unsafe program estimate and shows the amount of pessimism introduced when modifying the programs.

The serial method overestimates the WCET by at least a factor of 2. This is expected and is a result of our assumed timing model. However, for *DES* and *compress*, additional sources contribute. In *DES*, the small additional overestimation is due to data accesses with an unknown reference address. These unpredictable accesses must not be cached in order to keep the cache state predictable. This is accomplished by mapping the accessed data structures into a non-cacheable part of the memory. Then, unpredictable accesses will not interfere with the cache and will always

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

Table 6.1: The estimated WCET using the serial method and when using modified programs.

cause a cache miss. In *compress*, a small part of the overestimation is also due to unpredictable data accesses. In addition to this, the path analysis fails to eliminate all infeasible paths due to a pessimistic upper bound on a loop (a more thorough description of this loop can be found in Chapter 5).

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

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

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

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

In summary, our program modification method can perform well in conjunction with our symbolic execution method for all our benchmark programs. It works especially well for programs that have few variable-latency instructions and only one feasible path so that merging is avoided when analyzing the program. On the other hand, if a program contains

many variable-latency instructions or many feasible paths then the serial method could perform nearly as well or maybe better. For example, if optimization is enabled when compiling *matmult*, the variable-latency multiplications becomes relatively more frequent. This would change the slowdown from a factor of 1.2 to approximately 1.5, thus approaching the slowdown of the serial method.

6.6 Preemptive and non-preemptive scheduling

The WCET of a program is typically defined as the uninterrupted program execution time. Nonetheless, most systems use preemptive task scheduling which means that tasks can be interrupted at any time to allow an other high-priority tasks to run. This does not invalidate the WCET estimate of the high-priority task but when the original task continues execution, the timing state has typically been changed by the high-priority task in a way the can extend the WCET of the original program. This effect is typically accounted for in a schedulability analysis by adding an additional preemption delay covering these extrinsic timing effects between different tasks. Several published methods can calculate cache and pipeline related preemption delays. An good overview can be found in [Sch00] where Schneider shows that WCET analysis and preemption delay calculation can be more or less integrated.

When using the program modification method to handle unbounded mechanisms, the timing state in the system must be well defined at all points the program. An important consequence of this is that we must statically account for all unknown events. This forbids the use of preemptive scheduling where a program can be interrupted at any time. However, limited preemption would be possible by treating preemption points in the program as being similar to merge points. The timing state must be predictable at all points, regardless of the program being preempted or not. A further complication is that a typical computer system often include several other mechanisms that have an unpredictable nature. For example, techniques like DRAM refresh and DMA transfer of data can make the memory access time vary, which in turn would give all store and load instructions that experiences a cache miss an unpredictable latency. These features must also be accounted for if we want to use the program modification method.

To conclude, if we rely on preemptive scheduling or a system with other unpredictable mechanisms, we have only one option left. Namely to model all mechanisms using pessimistic but bounded timing models. For example, the serial-execution method does not rely on making unknown events safe and can be used together with preemptive scheduling.

Chapter 7

Data Cache Analysis

Data caching has proven to be one of the more complex techniques to analyze. Besides the symbolic execution method presented in this thesis, several other estimation methods have been proposed that predict the impact of caches on the WCET [BN94, FW98, LBJ+94, KMH96, HWH95, LMW95, LMW96, Mue97, OS97, TF98, WMH+97, WE00]. One general observation made from previous research is that the fetching of instructions is often highly predictable and can be accurately analyzed. On the contrary, memory accesses going to the data cache are often unpredictable and difficult to analyze. The reason for this is that a single load or store instruction can generate many different reference addresses and that these addresses also can depend on unknown input data.

In this and the next chapter, we will study how memory accesses to program data structures determine the success of estimating the WCET for a system that uses data caching. A conservative approach would be to assume that all memory accesses miss in the cache. Obviously, this is often overly pessimistic. A more successful approach should lead to no or a small overestimation of the WCET and we will present several methods that can successfully be used to analyze the impact of memory accesses on data caching.

The symbolic execution method presented in this thesis naturally analyzes memory accesses because it can take into account operands whose values are either known or *unknown*. Therefore, cache behavior of accesses whose reference addresses are known during analysis is completely predictable. On the other hand, accesses whose reference addresses are

unknown, either as a result of *unknown* input data or as a result of operand values that become *unknown* during the analysis, will result in unpredictable cache behavior. We will use this property to develop a method that can distinguish between predictable and unpredictable data accesses, and, extending it further, distinguish between predictable and unpredictable data structures.

In this chapter, we start in Section 7.1 to state concepts to reason about data cache predictability. We then present an approach for data cache analysis in Section 7.2 which is based on the notion of distinguishing between predictable and unpredictable data structures. Based on this, in Section 7.3, we build intuition into what types of data structures are expected to be predictable. We will test this intuition experimentally in Section 7.4 before we compare with related work in the area of data cache analysis in Section 7.5.

7.1 Data cache predictability definitions

To understand how the symbolic execution method can be extended to handle unpredictable accesses, we need to distinguish between the following: a data memory access instruction in the program code, the actual memory access done by the instruction at some point in time, and the data structure in the main memory that is the target of the access.

Definition 7.1 *An unpredictable memory access is a load or store access whose reference address is unknown during estimation of the WCET. Conversely, a predictable memory access is a load or store access whose reference address is known during the estimation of the WCET.*

The reasons why the reference address is unknown are twofold: First, it could be unknown because it depends on unknown input data. Second, even if the reference address does not depend on unknown input data, the WCET method could introduce uncertainties that actually make some of the operands in the program unknown. For example, in our method, the system state of two paths that meet is compared. If the same operand has different values in the two paths, it will be assumed to be unknown in order to make it possible to drop one of the paths and make sure that a worst-case value has been assumed. Therefore, whether an access is predictable or not depends also on the WCET estimation method used.

Definition 7.2 *An unpredictable memory access instruction is a load or store instruction that generates at least one unpredictable memory access. Conversely, a predictable memory access instruction is a load or store instruction that only generates predictable memory accesses.*

Because loop constructs are common in most programs, a particular load or a store instruction is executed many times. The purpose of this definition is to identify all load and store instructions that can generate at least one unpredictable memory access. One way of exploiting the fact that a memory access instruction is unpredictable would be to tag it as non-cacheable, thereby avoiding it to interfere with data that can be predictably cached.

Definition 7.3 *An unpredictable data structure is a data structure that is accessed by at least one unpredictable memory access. Conversely, A predictable data structure is a data structure that is accessed by only predictable memory accesses.*

This definition illustrates an approach to achieve a high predictable data cache performance. If a data structure is only accessed by loads and stores whose reference addresses are known (predictable) during the WCET estimation, the data structure can be predictably cached. Thus, if it is possible to automatically distinguish between predictable and unpredictable data structures, it should be possible to achieve a predictable data cache behavior. The method presented in the next section has this goal in mind.

The most important characteristic of unpredictable memory accesses is that we connect it to the WCET estimation method used. This is reasonable since it is hard to give a general, method-invariant definition of unpredictable accesses that is not very vague and thereby useless. An implication of the definitions is that whether a data structure is unpredictable or not also depends on the method used. In the following, when unpredictable data structures are mentioned, they are connected to the symbolic execution method presented in this thesis.

When looking at a fixed path through a program, predictable memory access instructions are typically those that always generate the same reference address or the same sequence of addresses, regardless of program input data. For example, many estimation methods [LMW96, KMH96, WMH+97, FW98, OS97] are capable of handling

accesses to global scalar variables. These variables are accessed using a fixed reference address that is independent of input data and is fairly easily deduced by an estimation method. Thus, these variables are predictable data structures when regarding all methods.

7.2 A predictable cache analysis method

We will now describe a method that makes it possible to identify and handle unpredictable memory accesses. We especially study how to apply this to the method of symbolic execution, but the principles presented here can be applied also to other WCET estimation methods presented in the literature as we will discuss in Section 7.5.

As a first step, we identify all unpredictable accesses by performing a WCET analysis. All memory accesses which have an unknown reference address can be collected. To reduce the amount of information all unknown references are not collected but only the corresponding memory access instructions. Thus, the result is a list of unpredictable memory access instructions and we consider all memory accesses from these instructions as unpredictable. In the next step, each unpredictable memory access instruction is connected to the corresponding data structure. This requires information from the compiler or user about which data structure each memory access instruction can touch. Each data structure that is accessed by an unpredictable memory access instruction is marked as unpredictable. In the final step, linking is redone in order to map all data structures marked as unpredictable into a special memory area. This memory area will be marked as non-cacheable.

When all unpredictable data structures have been identified and properly mapped, a final estimation of the WCET can be done. Now, the time taken to access an unpredictable structure is equal to the memory access time for a memory word. Assuming a memory hierarchy with a single level, the miss penalty is simply the memory access time. However, the time to access and transfer a memory block to the cache can be greater that the time to access and transfer a single word. Therefore, the miss penalty can sometimes be greater than the memory access time for a single word. If we would have allowed caching of the unpredictable access, the time charged could in the worst-case have been two cache miss penalties: one for the potential cache miss and one for the possibility of

replacing a useful block. In summary, by letting all unpredictable accesses bypass the cache we may gain more than a factor of two on the worst-case performance of data caches.

The approach presented here requires support in the computer system for controlling the caching of different memory areas. Fortunately, hardware mechanisms that permit us to do this exist in many microprocessors that use caches. In some systems, we can even choose if we want to cache or not to cache each individual data access. As an example, PowerPC 403 GA [IBM] has a double-mapped memory address space. This means that one physical memory location can be reached from two different addresses and we can choose different cacheability for the two addresses, one cache-enabled address and one cache-disabled address. Other hardware mechanisms also exist; in many general-purpose processors, support often exists for virtual memory where cacheability can be controlled on a per page basis. This can be used to create a similar double-mapping, allowing individual accesses to be either cached or not cached.

Even if hardware permits cacheability control of each individual memory access, this can be hard to exploit since it would require quite complex compiler support. Another available approach is to control each individual memory access instruction. This would allow caching of accesses to a data structure when it is used predictably in some parts of the program and not caching accesses when it is used unpredictably in other parts. Still, compiler interaction is needed and care must be taken to keep the contents of memory and cache consistent by flushing the cache at proper points or by using a write-through policy.

For simplicity, the evaluation has been limited to the control of cacheability on a data-structure level. This requires only support from the linker in order to control the placement of individual data structures. Also, there is no consistency problem as data is either cached or not cached.

7.3 Data structure classification

Based on the notion of predictable versus unpredictable data structures, we now develop intuition into what data structures are expected to fall into each of these two categories. It should be noted though that this generally depends on the WCET method used. We will therefore comment

Storage type	Explanation
Global	Global or static structures.
Stack	Stack allocated structures.
Heap	Dynamically allocated structures on the heap.

Access type	Explanation
Scalar	Only one element.
Regular array	Array accessed by regular, stride accesses.
Irregular	Irregular accesses but still input data independent.
Input dependent	Reference addresses directly depends on input data.

Table 7.1: Data structure classification based on storage type (upper) and access type (lower).

on to what extent the intuition depends on a certain WCET method.

In Table 7.1 we classify data structures with respect to storage type (upper table) and access type (lower table). The storage type determines what base address is used to access an element in the data structure. This base address is typically stored in a specific register (global, stack, or heap). Additionally, the access type determines how elements using the same base address are accessed and typically uses a fixed offset (scalar) or a varying offset that is calculated for each access (regular, irregular or input data dependent). Whether a data structure is predictable or not typically depends on both the storage type and the access type.

Let us first discuss predictability properties of data structures of different storage types. Global storage is the most simple one because the base address is a fixed known value produced by the linker. Stack allocated structures may look hard to analyze. Yet, they are often handled predictably by all estimation methods. The general approach adopted is to only estimate the WCET for complete programs since then the stack pointer is known. If only a single function or procedure in the program is analyzed the stack pointer would be unknown. The estimation method must keep track of the function call stack, and if a function is called from several places in the program each invocation of the function must be treated as a separate instance. Then, each function instance will have a fixed stack pointer value and thereby a known base address for stack

allocated structures. Our WCET estimation method are designed to estimate the WCET of complete programs where the stack pointer is known. Nevertheless, we will in the next chapter, see how to extend the symbolic execution method to handle the analysis of isolated functions where the stack pointer is unknown.

The most tricky storage type is heap storage, i.e., dynamically allocated memory on the heap. Due to its unpredictable behavior, this kind of allocation is not always permitted in real-time systems. However, our method allows some limited use of dynamic allocation. A necessary condition for predictable dynamic allocation is that the memory must be allocated in an order and in an amount that is independent of input data. This means that at each point in time, we always know which objects reside on the heap and in which order they are allocated. Then, the base address of the allocated data structures are known. Using dynamic allocation in this controlled manner may seem a bit pointless but can still be useful. For example, it would allow the programmer to write programs that reuse memory in a straight-forward way.

A known base address of a data structure is not enough to make it predictable. The access type must also be taken into consideration. For scalar variables, the base address is the only thing used. They are therefore predictable whenever the base address is predictable. Regular array accesses, i.e., accesses with a constant stride, are considered to be such accesses that can be predicted using e.g. data dependency analysis. This kind of analysis is treated in for example [FW98] and typically handles the case where the reference address is a simple function of the loop iteration variables. Many methods can analyze this kind of accesses.

A data structure accessed by irregular accesses may in theory be predictable since the accesses are independent of input data. However, many estimation methods would classify it as unpredictable because of lacking analysis of complex data dependencies. Yet, as we will see in Section 7.4, our estimation method manages to handle a case of irregular accesses, showing that some irregular data structures can indeed be predictable.

Finally, a data structure accessed by input data dependent accesses will always be unpredictable as long as input data is considered to be unknown.

In summary, we can note that many types of data structures are expected to be predictable using state of the art WCET estimation methods. We test this hypothesis in the next section.

7.4 Experimental results

We will now evaluate how well data caching can be analyzed in terms of the overestimation of the WCET. This also corresponds to the question of how much the worst-case performance is improved when using data caching. Data memory accesses have been classified based on a WCET estimation of seven benchmark programs and the fraction of memory accesses that go to unpredictable data structures has been determined. Furthermore, the data cache hit-rates and the corresponding WCET when counting all accesses to unpredictable data structures as misses have been measured.

7.4.1 Methodology

The symbolic execution method has been used to first classify all data structures according to the procedure presented in Section 7.2. Then, the WCET estimation tool has been extended to also collect all data memory accesses along the worst-case path during the analysis. The collected memory accesses have been classified according to which type of data structure they access. The categories of data structures considered are the ones presented in Section 7.3.

The timing model assumed for the hit-rate and WCET estimations represents an ideal architecture containing only a data cache. All instructions execute in a single cycle except on a data cache miss when a miss penalty of 10 cycles is added. The data cache is a 2 Kbyte direct-mapped cache with 16-byte block size. When regarding the data cache, loads and stores were treated as equivalent to each other.

An overview of the seven programs can be seen in Table 7.2. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *jfdctint*, *DES* and *compress*. The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the programs. No optimization was enabled except for *DES-opt* which was compiled with the option `-O2`. The simulated run-time environment contained no operating system; consequently, we disabled all calls to system functions such as I/O in the programs.

Name	Description
matmult	Multiplies two 50x50 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
DES	Encrypts 64-bit data
DES-opt	DES compiled with optimizations enabled
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

Table 7.2: Characteristics of the programs used.

7.4.2 Results

The results from the memory access classification can be seen in Figure 7.1. The first five benchmarks, *matmult*, *bsort*, *isort*, *fib*, and *jfdctint*, were found to contain only predictable data structures. In all these five benchmarks, the majority of the accesses aimed at scalar variables allocated on the stack. Another common effect is that most regular array accesses were either allocated globally (*matmult*, *bsort*, and *jfdctint*) or on the stack (*isort*). All the accesses to these data structures were found to be predictable when estimating the WCET and we thereby confirm the intuition from Section 7.3 that scalar and regular array accesses to global or stack allocated structures are predictable.

In the last benchmarks, *DES*, *DES-opt*, and *compress*, unpredictable data structures were found. In *DES* two arrays are read using an index that depends on unknown input data. However, only 0.6 % of all accesses went to these arrays and the majority of accesses did again reference scalar variables allocated on the stack. The same is valid for *DES-opt*, although the amount of scalar stack accesses has been reduced. This reduction is expected and can be explained by the fact that scalar stack storage is often used as temporary storage by the compiler. Part of this temporary storage can often be eliminated by the compiler when enabling optimization. The reduction of scalar stack accesses makes the other

	matmult	**bsort**	**isort**
Scalar, global	1.4 %	0.1 %	2.6 %
Scalar, stack	49.9 %	77.0 %	66.6 %
Regular, global	48.7 %	22.9 %	
Regular, stack			30.8 %
Total accesses	1057597	130652	876
	fib	**jfdctint**	
Scalar, global	3.5 %	0.5 %	
Scalar, stack	96.5 %	80.9 %	
Regular, global		18.6 %	
Total accesses	397	2754	
	DES	**DES-opt**	**compress**
Scalar, global	4.1 %	10.3 %	24.4 %
Scalar, stack	84.5 %	59.2 %	36.4 %
Regular, global	5.4 %	14.5 %	1.7 %
Regular, stack	0.6 %	1.5 %	
Irregular, global	4.9 %	13.0 %	
Input dep, global	0.6 %	1.5 %	37.5 %
Total accesses	45876	17200	8852

Figure 7.1: Classified memory accesses from the worst-case program path.

Name	Fraction predictable	Data cache hit ratio	WCET	No data cache WCET	No data cache Ratio
matmult	100 %	92.1 %	7899863	17639883	2.2
bsort	100 %	97.8 %	320887	1598547	5.0
isort	100 %	98.1 %	2765	11355	4.1
fib	100 %	96.5 %	838	4668	5.6
jfdctint	100 %	98.7 %	6361	33551	5.3
DES	99.4 %	98.8 %	124276	577436	4.7
DES-opt	98.5 %	96.8 %	53905	220345	4.1
compress	62.5 %	62.0 %	82190	137050	1.7

Table 7.3: Fraction predictable accesses and corresponding hit-rate.

accesses relatively more significant. Still, only 1.5 % of all accesses is unpredictable in *DES-opt*.

An interesting fact is that the WCET method managed to handle some cases of irregular arrays accesses in *DES* and *DES-opt*. Some arrays were accessed using an index obtained from another, regular array access. Thus, the array references are independent of input data and were found to be predictable by the WCET estimation method we used. This kind of array accessing would have been found unpredictable by many other WCET estimation methods.

In the final benchmark, *compress*, four data structures were found to be unpredictable. The dominant structures in this case were two hash tables (total size 1.5 Kbyte) indexed by unknown input data. In total, 37.5 % of all accesses went to these unpredictable data structures. Of these 37.5, unpredictable accesses contributed with 30. The rest, 7.5 was the contribution from predictable accesses during the initialization of the hash tables. This means that it would have been better to control the cacheability on the instruction level instead of a data structure level. However, the gain would have been small.

Table 7.3 summarizes the amount of memory accesses that were found to be predictable. Also shown is the corresponding data cache hit ratio. As explained before, almost all accesses were predictable in all benchmarks except in *compress*. This can also be seen in the hit ratio numbers which are close to 100 % in all benchmarks except *compress* where the unpredictable accesses cause a significant reduction of the hit ratio.

To understand the importance of data caching, we have also included the estimated WCET for two cases: one when all accesses to predictable data structures are cached and another when caching is disabled and all accesses are counted as misses. Ratio is the cache-disabled WCET divided by the cache-enabled WCET. It represents the improvement of the worst-case performance obtained when including a data cache. Clearly, for the majority of the programs studied, data caching is efficient and improves the worst-case performance significantly, in many cases by a factor of four or more. Even for *compress*, we get a considerable improvement by a factor of 1.7 when using a data cache in spite of all the unpredictable accesses encountered.

7.5 Discussion and related work

The symbolic execution method handles quite complex data dependencies and identifies many data structures as predictable, even the irregular ones in *DES* as seen from the results. An interesting question is if the use of another WCET estimation method would have changed the results drastically. Probably, other methods would also make a good analysis of many of the programs. For example, the methods presented in [LMW96, KMH96, WMH+97, FW98, WE00] should be able to produce similar results as the symbolic execution method for the first five benchmarks. These benchmarks only contain data structures that are fairly simple to handle. The most complex one is regular array accesses which probably could be handled by all mentioned methods. On the other hand, the other methods would probably perform worse on *DES* and *DES-opt*, due to the irregular array accesses present, which are accurately analyzed by the symbolic execution method. An exception to this might be the method by Wolf and Ernst [WE00]. They simulate the parts of a program that only contain a single feasible path. If they successfully identify *DES* or *DES-opt* as containing only a single feasible path, then they would probably get similar results as obtained here.

The handling of *compress* by other methods is largely dependent on which strategy they use to handle unpredictable accesses. This strategy is not always made clear from the descriptions of the methods but in for example [KMH96] they adopt the strategy of caching unpredictable accesses. Then, to make a safe estimate of the WCET, two miss penal-

ties must be added for each unpredictable access: one for the possibility of a cache miss and another for the possibility of replacing another useful block. This strategy would reduce the worst-case performance quite drastically for *compress*. The resulting estimated WCET by our method when using this strategy is 108690 cycles which is only a factor of 1.26 better than not caching at all. This can be compared to a factor of 1.7 obtained when using the method in this paper.

The method of identifying unpredictable data structures, described in Section 7.2, is not limited to the symbolic execution method used in this thesis. The same method could be used together with other estimation methods. The important criterion is that a list of all instructions generating accesses with an unknown reference address can be created during the estimation. This information can probably be generated early in the estimation procedure since deducing the reference addresses of data accesses is often the first step needed when analyzing data cache behavior.

The results obtained by the use of the symbolic execution method are probably very hard to improve upon since the unpredictable accesses in for example *compress* depends on unknown input data and are very random in nature. Thus, the results obtained for the benchmarks is as good as it can get. However, for other programs it is possible that accesses to data structures that our method would classify as unpredictable could be turned into predictable when using other methods. As an example, consider regular accesses to an array allocated on a heap with an unknown base address. The WCET method we use would then classify the array as unpredictable. However, another method could predict some of the behavior based on upper or lower bounds on the number of misses or hits in the cache. The feasibility of this approach depends on how advanced data dependency analysis that can be made and we find it currently unclear if other methods could handle a case like this. For predictable data structures, many methods [LMW96, KMH96, WMH+97, FW98] use the approach of calculating upper and lower bounds on the number of misses or hits.

A rather different approach of handling data cache analysis is described by Basumallick and Nilsen [BN94] where they use a register allocation algorithm to allocate data into different cache blocks. In this way, they arrange the data in a way that guarantees a certain number of cache hits. However, the same problem with unpredictable accesses will still be present.

The focus of this chapter has been to obtain tight WCET estimations for a system with a data cache. If tight WCET estimation is not of primary concern it may be more fruitful to cache also unpredictable data since this probably reduces the average execution time. Then, to make a safe estimate, the approach taken by [KMH96] must be adopted which means that two miss penalties must be added for each unpredictable access. Hence, it is possible to trade tightness of WCET estimation for increased average performance.

Chapter 8

Unknown Data Placement

In this chapter, we continue with the problem of data cache analysis but in a new context. Namely, the context of estimating the WCET for a single subroutine. This is useful when WCET analysis is needed before a program is fully written. For example, the subroutine could be a library function that will be used by several other programs. Furthermore, the analysis of a single subroutine can also be useful to reduce the complexity when analyzing a complete program. When analyzing a single subroutine, several new sources of overestimation appears. For example, the WCET estimate of a subroutine A that calls a subroutine B will typically be tighter if A and B are analyzed together instead of first analyzing subroutine B and then use the result when analyzing A. However, in this chapter, we will only study ways to reduce the overestimation possible from the use of data caching.

The overestimation from data caching is determined by the possibility of deciding the target addresses of memory accesses. Now, the problem is that when considering a single subroutine it is a common case that the target address of memory accesses will be unknown due to its dependency on unknown input. Typically, the initial stack pointer and input parameters like pointers to data objects in memory will be unknown. As we saw in the previous chapter, this would normally make all memory accesses unknown, leading to overestimation of the WCET. Fortunately, even if the placement of stack and data objects is unknown, the accesses are still somewhat predictable.

In Section 8.2, we present a new conflict analysis method that ex-

tends the symbolic execution method presented in this thesis and that makes it possible to reduce the over-estimation of the WCET due to the unknown memory accesses. This extension makes it possible to analyze the interaction of different memory accesses, such as global data accesses, interacting and conflicting with stack accesses. It exploits the fact that although the base address used for a sequence of accesses may depend on unknown input data, the offsets relative to this base address may be known, making it possible to better predict the cache behavior.

The experimental results, which can be found in Section 8.3, show that the conflict analysis method in many cases finds tight estimates of the worst-case number of data cache misses. Also, the time complexity of the analysis is found to be reasonably low. The results show that the use of a traditional data cache can in many cases lead to a predictable estimate of the WCET even when analyzing a single subroutine.

Previously published data cache analysis methods [FW98, KMH96, LMW96, WMH+97, WE00] can also be used to handle access sequences with unknown base addresses. However, they are only able to handle a single such sequence and cannot analyze the interaction between many sequences. Thus, for subroutines containing several access sequences with an unknown base address, these methods may grossly over-estimate the WCET. We discuss this issue further in Section 8.4.

8.1 The problem

To better understand the problem with conflicting accesses in a data cache, we will start by looking at a simple example. In the example and in the rest of this chapter we will use the following system model.

8.1.1 System model

For the purpose of this chapter, we focus only on the effect of a data cache on the execution time. We treat all memory reads and writes as equivalent in terms of cache effect. The data cache is assumed to be a traditional cache where placement is managed by hardware. The least recently used block in the set (true LRU) is replaced when a new block is inserted. Furthermore, it is assumed that one or several regions of memory can be marked as being non-cacheable. To simplify the presentation we require that all initial pointer values are aligned with the data

```
matmult(matrix A, B, R)
{
  int x, y, z;

  for (x = 0; x < 10; x++)
    for (y = 0; y < 10; y++) {
      R[x][y] = 0;
      for (z = 0; z < 10; z++)
        R[x][y] += A[x][z] * B[z][y];
    }
}
```

Figure 8.1: Matrix multiplication subroutine

cache block boundaries. Unless otherwise stated, we assume a 2048 bytes, direct-mapped cache with a block size of 16 bytes.

8.1.2 Conflicting accesses

Let us assume we want to find an upper bound on the number of data cache misses occurring when the matrix multiplication subroutine in Figure 8.1 is run. The upper bound must take into account that the matrices can be placed at arbitrary locations in memory and that the value of the initial stack pointer is unknown. The subroutine multiplies two 10 x 10 matrices, A and B, and puts the result in matrix R. The `matrix` type is defined as a pointer to an array of integers. When compiled (without optimization), the local variables `x`, `y`, and `z` are allocated on the stack. This means that we will have a mix of accesses going either to the stack area or to the different matrices allocated somewhere in memory.

Figure 8.2 shows the behavior of this subroutine regarding data cache misses. The number of data cache misses has been measured for 1000 random placements of data, and then sorted. The minimum number of data cache misses occurring is 78. This represents the number of cold misses and matches approximately the working set of the subroutine, which is $1200/16 = 75$ blocks for the three matrices. Since the cache capacity is higher than 75 blocks we have no capacity misses and the extra cache misses we see for the majority of all test samples are due to conflicting mapping of data into the data cache. The worst observed combination of input pointer values resulted in 1000 cache misses.

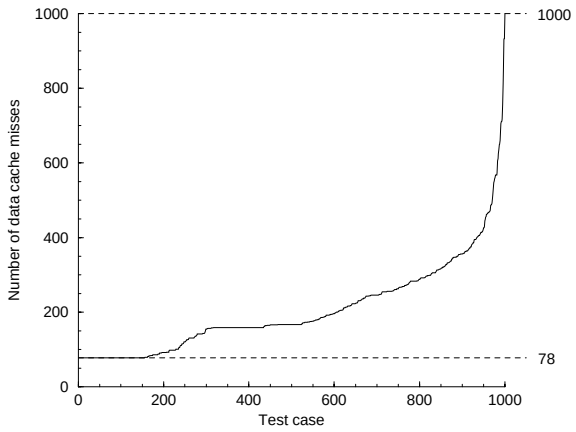


Figure 8.2: Measured number of cache misses.

This example shows the importance of testing all input values in order to find the worst case. However, an exhaustive test of all different input pointer values is often not practical. A pointer to a data object determines in which cache set the first block of the data object will be cached. Thus, for block aligned pointers, the number of different cache mappings is equal to the number of cache sets, *numsets*. If a subroutine depends on s unknown pointers, then the number of combinations to test is:

$$combinations = numsets^s$$

In the example, $numsets = 2048/16 = 128$. Thus, four pointers give us $128^4 \approx 268$ million combinations to test. The interesting question is if it is possible to find the worst case without doing exhaustive testing. As will be demonstrated in the following sections this is indeed possible and in Section 8.3, we find that the actual worst-case for this subroutine is not 1000 but 1252.

8.2 Approach

As demonstrated in Chapter 7, the symbolic execution method is a powerful method to use for data cache analysis. However, data structures with

an unknown storage location were always classified as unpredictable. In the following sections, we will show how to extend the capability of this method to identify such data structures as predictable despite the storage location being unknown.

In Section 8.2.1, we will first explain how to identify data structures that are predictable but have an unknown placement. Then, in Sections 8.2.2, 8.2.3, and 8.2.4, we will explain how to use this information in a data cache analysis without resorting to exhaustive testing of all possible combinations of different placements. Finally, in Section 8.2.5, we will show how the new data cache analysis handles multiple paths and how it affects the merge operation.

8.2.1 Data structure identification

The first step is to identify data structures that are accessed in a predictable manner but depends on an unknown base pointer that points to the location in memory where the data structure is allocated.

When analyzing a single subroutine, like the one in the example in Figure 8.1, we first assign unknown values to all unknown input values at the start of the analysis. Using our simulator tool we then obtain a memory access trace for each path in the program, identifying all accesses going to data structures with unknown placement as unknown. The data structures we are looking for will be classified as unpredictable.

Among the unpredictable data structures we can then use a testing approach to identify the ones that can be turned into predictable ones. The approach is to assign arbitrary values to all input data dependent base pointers. This is done by simply linking the subroutine with a small test program that allocates the data structures to some place in memory and then call the subroutine. Then, we run our simulator tool again to see if any data structure previously classified as unpredictable gets classified as predictable. These data structures are marked as being *predictable with an unknown placement*.

After the identification is done, all data structures have been classified as either *predictable* if the access pattern is independent of input data, *predictable with an unknown placement* if the access pattern is independent of input data but the storage location depends on input data, or *unpredictable* if the access pattern may depend on input data regardless of the storage location being known or unknown.

Finally, to make a WCET estimation, all predictable data structures are allocated to some arbitrary memory location. To make a correct analysis, the data cache analysis presented in Chapter 7 is extended with a conflict analysis to take the unknown placement into account.

8.2.2 Data cache analysis

After the identification step, the data cache analysis is responsible for analyzing the memory accesses produced by the simulator. Whenever a load or store instruction is executed, we must determine if it hits or misses in the cache. The added complexity in this case is that we want to find out if the access can result in a cache hit regardless of the placement of data structures.

To be able to analyze the interaction of different memory accesses we keep a history of all memory accesses that have occurred. Whenever a new access occurs we add it to the end of the history list and also do a local analysis to determine if it hits or misses in the cache. An example of such a history list can be seen in Table 8.1. The first column in the table shows the accesses made by the program expressed as offsets from the initial stack pointer or from the pointer to data structure A. In the beginning of the analysis, fixed values are assigned to these pointers ($sp = 100000, p_A = 3800$) and columns 2 and 3 in the table show the content of the list of accesses used during the analysis. In the list, each access is labeled according to the data structure it targets. The memory accesses belonging to a certain data structure are defined as being part of the same *memory access sequence*.

To decide if an access is a hit or a miss, we first imagine that each access sequence uses its own separate cache. This makes it possible to identify compulsory misses such as cold misses or misses due to internal conflicts, i.e., conflicts with other accesses in the same sequence. If an access is not identified as a miss in this step, we continue with an analysis where we take into account possible conflicts from other sequences. Accesses from other sequences are then assumed to be interfering in the most pessimistic manner.

An example of the result of an analysis, for both a direct-mapped and a 2-way set associative cache, can be seen in Table 8.1. For the first 6 accesses, each access can be classified by only looking at accesses belonging to the same access sequence. Thus, the analysis is similar to a traditional

	History of accesses		Analysis, direct-mapped		Analysis, 2-way assoc.	
	Address	Sequence	Set #	Hit/Miss analysis	Set #	Hit/Miss analysis
1	sp - 109c	fef64 stack	118	Miss (cold)	54	Miss (cold)
2	sp - 10a0	fef60 stack	118	Hit same block as 1	54	Hit
3	sp - 10a4	fef5c stack	117	Miss (cold)	53	Miss (cold)
4	$p_A + 850$	4050 A	5	Miss (cold)	5	Miss (cold)
5	$p_A + 50$	3850 A	5	Miss (cold)	5	Miss (cold)
6	$p_A + 450$	4050 A	5	Miss conflict from 5	5	Hit
7	sp - 10a0	fef60 stack	118	Miss if conflict from 4,5,6	54	Miss if conflict
8	sp - 10a4	fef5c stack	117	Miss if conflict from 4,5,6	53	Miss if conflict
9	$p_A + 450$	4050 A	5	Miss if conflict from 7,8	5	Hit

Initial stack pointer: $sp = 100000$, initial pointer to data structure A: $p_A = 3800$
Size of data cache: 2048 bytes. Block size: 16 bytes.

Table 8.1: Example of basic analysis for a direct-mapped and a 2-way cache.

cache analysis. However, for accesses 7,8, and 9, each access can miss due to conflicts with accesses belonging to other sequences. When analyzing access 7, we find that this access must be assumed to be a miss since p_A can be set to a value that makes accesses 4,5, and 6 map to the same cache set as access 7. This is true for both the direct-mapped and the 2-way cache since accesses 4,5, and 6 will replace two blocks in a cache set. Access 8 is analyzed in the same manner and must also be assumed to be a miss. The last access, access 9, is more interesting. Here, the stack pointer can be set to a value that makes accesses 7 and 8 map to the same cache set as access 9. However, the stack accesses can only replace at most one block in a cache set, so we get a possible conflict miss for the direct-mapped cache but a guaranteed hit for the 2-way cache.

An interesting detail in the example concerns access 7 and 8. If we study the example carefully, we actually find that only one of the accesses 7 and 8 will miss. The stack access cannot replace both accesses simultaneously, only one of them. In the next sections, we will first present a basic algorithm that cannot discover this case. Then, we will show how to extend the basic algorithm in order to improve the accuracy of the analysis. Using the extended version of the algorithm we can indeed identify one of the accesses 7 and 8 as a hit.

8.2.3 Basic algorithm

The basic algorithm used can be seen in Figure 8.3. It analyzes a new access A with target address r. First, the list containing all previous accesses is searched, starting from the end, to find an access targeting the same sequence and memory block, *memblock*, as access A. If no such access is found, the access is a cold miss.

If *memblock* is found at a position *pos*, we must analyze all intermediate memory accesses and see if they can replace *memblock*. In this step, we sort all intermediately accessed memory blocks into different sets according to the sequence and cache set they map to. This sorting can be seen as a kind of backwards cache simulation where each sequence is stored in a separate cache. Starting with the last access, the number of the accessed memory block is put into a set $blocks(q, i)$, where q is the sequence that the access belongs to and i is the cache set it maps to. This is done for all accesses until *pos* is reached.

To see if *memblock* can be replaced, the algorithm calculates the max-

```
# r is the target address of the memory access A.
# seq is the sequence that the access belongs to.
# blocksize is the cache block size.
# assoc is the cache associativity.
# size is the cache size.
# numsets = size/blocksize/assoc is the number of cache sets.
# |set| = the number of elements in set set.

function Analyze(A)
  memblock = r/blocksize
  pos = ListSearchBackwards(memblock, seq)
  if memblock not found then
    MISS (cold)
  else
    blocks = SortAccesses(pos)
    cacheset = (r/blocksize) mod numsets
    age_max = 1
    age_max = age_max + |blocks(seq, cacheset)|
    for all sequences q ≠ seq do
      age_max = age_max + max_{0≤i<numsets} |blocks(q, i)|
    end for
    if age_max > assoc then
      MISS (possible conflict)
    else
      HIT (guaranteed)
    end if
  end if
end function

function SortAccesses(pos)
  return blocks where
    blocks(q, i) = set containing all different memory
      blocks from end of history list to position pos,
      of sequence q mapping to cache set i.
end function
```

Figure 8.3: Basic hit/miss analysis algorithm for a new access A with target address r.

imum possible age, age_{max}, of *memblock*. The age of a block is used by the least-recently used replacement scheme in order to replace the oldest block when a new block needs to be inserted. If age_{max} is greater than the associativity of the cache then *memblock* may have been replaced. The maximum age is found by counting the number of unique intermediate accesses mapping to the same cache set as access A. For accesses belonging to the same sequence as A, this is easily found as the number of elements in $blocks(seq, cacheset)$, where *seq* and *cacheset* are the sequence that A belongs to and the cache set that A maps to, respectively. For other sequences, any cache set can interfere. Therefore, for each other sequence, the maximum number of elements found in any cache set is picked. Finally, if $age_{max} > assoc$, access A is a possible miss, otherwise it is a guaranteed hit.

8.2.4 Improving accuracy

When the basic algorithm analyzes each access of a sequence *seq*, it uses the pessimistic simplification that each sequence other than *seq* causes the maximum possible interference. However, by doing this analysis locally for each access the algorithm derives quite a pessimistic view of the conditions necessary for conflicts to occur.

Ideally, we want to calculate the maximum number of misses among all accesses that can occur considering all possible placements of data structures (an exhaustive algorithm). To express this more formally we first need some definitions. The placement of a data structure is determined by its corresponding initial pointer value. However, the cache behavior is only dependent on where the data is mapped in the cache memory. Therefore, to consider all placements of data we need only consider all offsets, o_i, of a sequence i in the cache. Since all pointers are assumed to be aligned with data cache block boundaries, the offset can be defined as the index of the cache set to which the initial pointer maps, i.e., $o_i \in S$ where $S = [0, numsets - 1]$ and *numsets* is the number of cache sets. Let $\mathbf{o} = (o_1, o_2, \ldots, o_{s-1}) \in S^s$ be a vector of dimension s containing the offsets for all sequences where s is the number of sequences. The set $\mathbf{S} = S^s$ contains all possible combinations of placement for all data structures. We can now express the maximum number of misses in the ideal case as:

$$ideal = \max_{\mathbf{o}\in\mathbf{S}} \left(\sum_{0\le A<n_a} miss(A,\mathbf{o}) \right)$$

The sum is made over all accesses A in the program (along a single path, in Section 8.2.5 we show how multiple paths are handled) and n_a is the total number of accesses. The value of the function $miss(A,\mathbf{o})$ is 1 if access A may miss when data structures are placed according to offsets $\mathbf{o}$, otherwise the value is 0. This means that for each combination of placements we do a traditional cache analysis of all accesses and pick the combination that gives the maximum number of misses.

The basic algorithm, described in the previous section, go through all accesses and does a local analysis for each access. The local analysis includes the examination of all possible data structure placement offsets. It calculates:

$$\begin{aligned} basic &= \sum_{0\le A<n_a} \begin{cases} 1 & \text{if } Analyze(A) = \text{MISS} \\ 0 & \text{otherwise} \end{cases} \\ &= \sum_{0\le A<n_a} \left(\max_{\mathbf{o}\in\mathbf{S}} miss(A,\mathbf{o}) \right) \end{aligned}$$

The difference is that the max function has been moved inside the summation, making the analysis more pessimistic. Our approach to get improved accuracy is to do something in between the ideal and the basic algorithm. By splitting the basic analysis into different cases, we can do an exhaustive test of all cases and for each case do a basic analysis using a restricted set of offsets.

Let K be the number of cases to restrict each offset to. Then, for a given case $c_i \in C$, where $C = [0, K-1]$, we restrict offset o_i to:

$$o_i = Kh_i + c_i$$

where: $h_i \in H$, and $H = [0, numsets/K - 1]$. For example, if $K = 2$ we restrict the offsets to even or odd values when setting $c_i = 0$ or $c_i = 1$, respectively. The number of cache sets, *numsets*, must be a multiple of K. Thus, K is typically a power of 2.

We can now formulate an expression for the estimated maximum number of misses that the improved algorithm calculates. First, we introduce the vector notations: $\mathbf{c} = (c_0, c_1, \ldots, c_{s-1}) \in \mathbf{C} = C^s$, and $\mathbf{h} = (h_0, h_1, \ldots, h_{s-1}) \in \mathbf{H} = H^s$. Then, the result from the improved algorithm can be expressed as:

$$\begin{aligned} improved &= \max_{\mathbf{c}\in\mathbf{C}} \sum_{0\le A<n_a} \left(\max_{\mathbf{h}\in\mathbf{H}} miss(A,\mathbf{o}) \right) \\ \mathbf{o} &= K\mathbf{h} + \mathbf{c} \end{aligned}$$

Thus, for a given case $\mathbf{c}$ we do a basic analysis but only include conflicts from other accesses if they occur for the restricted set of offsets. By setting $K = 1$ or $K = numsets$ the improved algorithm reduces to the basic or ideal algorithm, respectively.

In practice, it is possible to analyze all cases simultaneously by keeping a table, $miss\_table(\mathbf{c})$, that holds the possible number of misses discovered so far in the analysis for each case $\mathbf{c}$. Then, in the end, the result is obtained from:

$$improved = \max_{\mathbf{c}\in\mathbf{C}}(miss\_table(\mathbf{c}))$$

To better understand how the improved algorithm works, we will again study the example in Table 8.1 and see how accesses 7 and 8 is classified by the improved algorithm. The result from an improved analysis for a direct-mapped cache can be seen in Table 8.2. Each offset is split into 2 cases ($K = 2$) for a total of 4 cases since we have 2 sequences. Each case is given by the variables $c_{stack}, c_A \in [0, 1]$. As an example, for $c_{stack} = 0$ and $c_A = 1$ we restrict the offsets to:

$$o_{stack} = 2h_{stack} + 0$$

$$o_A = 2h_A + 1$$

Even offsets for the stack pointer means that access 7 maps to even cache sets and access 8 to odd cache sets. Odd offsets for the pointer to data structure A means that access 6 maps to even cache sets. If access 6 may map to the same cache set as accesses 7 or 8 we may get a conflict

	History of accesses			Possible cache set mappings for restricted offsets			
				$c_{stack} = 0$	$c_{stack} = 0$	$c_{stack} = 1$	$c_{stack} = 1$
	Address	Sequence	Set #	$c_A = 0$	$c_A = 1$	$c_A = 0$	$c_A = 1$
6	4050	A	5	1, 3, 5, ...	0, 2, 4, ...	1, 3, 5, ...	0, 2, 4, ...
7	fef60	stack	118	0, 2, 4, ...**h**	0, 2, 4, ...**m**	1, 3, 5, ...**m**	1, 3, 5, ...**h**
8	fef5c	stack	117	1, 3, 5, ...**m**	1, 3, 5, ...**h**	0, 2, 4, ...**h**	0, 2, 4, ...**m**

m= possible cache miss, **h**= cache hit.

Table 8.2: Example of improved analysis ($K = 2$) for a direct-mapped cache.

miss. Thus, when the possible cache set mappings overlap we may get conflict misses. The improved analysis manages, in this case, to find the fact that only one of the accesses 7 and 8 can miss due to conflicts with access 6 and we get at most one miss for these accesses for each case.

The improved algorithm can be seen in Figure 8.4. The case when all offsets $o_i = 0$ is defined as being the case when the original pointer offsets are used. In this case, the reference addresses found in the list of accesses are used unmodified. If $o_j = k$ for some j, it means that the accesses belonging to sequence j are translated k sets. In the algorithm, this translation affects the *SortAccesses* function. However, instead of building a new *blocks* table for each case, a translation of the index is done during the conflict analysis. The correctly translated $blocks_{case}(q, j) = blocks(q, i)$, where:

$$i = j - o_q(h_q) \pmod{numsets}$$

When calculating the maximum age of *memblock*, the algorithm must take into account that accesses in other sequences only conflict if they may map to the same cache set *cacheset* as *memblock* for a set of restricted offsets $o_i(h_i) = h_i K + c_i$. First, *cacheset* must be translated according to the offset of the sequence it belongs to, $o_{seq}(h_{seq})$. We get:

$$\begin{aligned} cacheset_{case} &= \\ &= cacheset + o_{seq}(h_{seq}) = \\ &= cacheset + h_{seq}K + c_{seq} \pmod{numsets} \end{aligned}$$

Then, the conflicting cache sets in a sequence q is found in $blocks_{case}(q, j)$, where $j = cacheset_{case}$. The index need to be translated to be used for $blocks(q, i)$. We get:

$$\begin{aligned} i &= cacheset_{case} - o_q(h_q) = \\ &= cacheset + h_{seq}K + c_{seq} - h_q K - c_q = \\ &= cacheset + h'K + c_{seq} - c_q = \\ &= h'K + set_q \pmod{numsets} \end{aligned}$$

```
# miss_table(c_0, c_1, ...) = the number of misses for each
#                             case so far.
# K = number of cases to split each offset into.

function AnalyzeImproved(A)
  memblock = r/blocksize
  pos = ListSearchBackwards(memblock, seq)
  if memblock not found then
    increment all entries in miss_table
  else
    blocks = SortAccesses(pos)
    cacheset = (r/blocksize) mod numsets
    age_max = 1
    age_max = age_max + |blocks(seq, cacheset)|
    for all cases 0 ≤ c_0, c_1, ... < K do
      age_case = age_max
      cacheset_case = cacheset + c_seq   (mod K)
      for all sequences q ≠ seq do
        set_q = cacheset_case − c_q   (mod K)
        age_case = age_case +
            max_{0≤h'<numsets/K} |blocks(q, h'K + set_q)|
      end for
      if age_case > assoc then
        increment miss_table(c_0, c_1, ...)
      end if
    end for
  end if
end function
```

Figure 8.4: Improved hit/miss analysis algorithm for a new access A with target address r.

In the algorithm, we let the index h' range from $0 \leq h' < numsets/K$ and $set_q = cacheset + c_{seq} - c_q \pmod K$.

As we will demonstrate in Section 8.3, the improved algorithm is very flexible. By choosing few cases we get a fast, less accurate algorithm. By increasing the number of cases we get a slower but more accurate algorithm. In Section 8.3, we evaluate this tradeoff.

8.2.5 Path analysis and merging

Up until now, we have only dealt with the problem of analyzing memory accesses from a single path in the program. To extend the analysis to multiple paths, we must integrate the analysis into the symbolic execution method. This is done by simply replacing the data cache analysis in the timing model introduced in Chapter 3 with the new conflict analysis. Instead of a data cache timing model with a state that consists of an array of tags, we now include the state of the conflict analysis, CA, that consists of the list of accesses, *alist*, and the table of misses, *miss_table*. However, since the conflict analysis can classify a memory access instruction as both a hit or miss depending on the case, the pipeline analysis should actually be done for both the cache hit and the cache miss cases. However, this could lead to an exponential growth of the number of pipeline states. The simple solution adopted here is to assume that all data accesses will hit in the cache when doing pipeline simulation and separately add the penalty for the data cache misses. The expression for the total estimated WCET introduced in Section 3.2.2 is now changed into:

$$\text{estimated } wcet = \max_{r \in R}(PL(r)) + \max_{\mathbf{c} \in \mathbf{C}}(miss\_table(\mathbf{c}))P_{DC} + pen$$

where the *miss_table* term has been added, denoting the maximum number of possible data cache misses which is multiplied by P_{DC}, the data cache miss penalty. As previously explained, PL is the timing state of the pipeline, R is the set of pipeline resources, and *pen* is the accumulated merge penalty.

The WCET expression above does not take into account the possibility of an overlap between a cache miss and some other long latency operation in the pipeline. However, this can be improved upon. For example, if an access is a cache miss for all cases, it can really be treated as a cache miss when doing the pipeline simulation. This kind of improvements has not been further studied.

Besides the change in the expression for the estimated WCET, we must also include the timing state of the conflict analysis in the expression for the upper bound $\Delta WCET$ introduced in Section 3.1. This is done by replacing the upper bound for the data cache, Δ_{DC}, with an upper bound for the conflict analysis. We introduce $\Delta_{conflict}(CA_A, CA_B)$ as being the upper bound on the difference in estimated WCET possible from the conflict analysis state CA_A when compared to CA_B. We now get:

$$\begin{aligned} \Delta WCET(t_A, t_B) &= \Delta_{pipeline}(PL_A, PL_B) + \\ &\quad \Delta_{IC}(IC_A, IC_B) + \\ &\quad \Delta_{conflict}(CA_A, CA_B) + \\ &\quad pen_A - pen_B \end{aligned}$$

where:

$$\begin{aligned} &\Delta_{conflict}(CA_A, CA_B) = \\ &\quad P_{DC}(\Delta_{alist}(alist_A, alist_B) + \Delta_{misstable}(miss\_table_A, miss\_table_B)) \end{aligned}$$

What remains is to define Δ_{alist} and $\Delta_{misstable}$. The result from the function $\Delta_{misstable}(miss\_table_A, miss\_table_B)$ should be an upper bound on the number of possible extra misses that $miss\_table_A$ can lead to that $miss\_table_B$ cannot lead to. This is simply the maximum difference for any case (entry) in the table. For example, assume that for one case the number of possible misses in $miss\_table_A$ is greater than the the number of possible misses for the same case in $miss\_table_B$. At the end of the analysis, this case may prove to be the maximum one and thus represents the final WCET. Thus, we get:

$$\begin{aligned} &\Delta_{misstable}(miss\_table_A, miss\_table_B) = \\ &\quad \max_{c_0, c_1, \ldots, c_{s-1}} (miss\_table_A(c_0, c_1, \ldots) - miss\_table_B(c_0, c_1, \ldots)) \end{aligned}$$

The upper bound $\Delta_{alist}(alist_A, alist_B)$ for the list of accesses is calculated by the algorithm in Figure 8.5. In principle, one extra cache miss is


```
# alist_A = list of accesses in path p_A.
# alist_B = list of accesses in path p_B.
# alist(last) = the last access in the list
# access_A = access_B ⇒ accesses refer to same memory block
#                        and belongs to same sequence.

function Δ_alist(alist_A, alist_B)
  remove superseded accesses in alist_A and alist_B

  while alist_A(last) = alist_B(last)
    remove last element in both lists
  end while

  return number of accesses left in alist_B
end function
```

Figure 8.5: Algorithm for calculating the upper bound Δ_{alist}.

possible for each access in the future that would be classified as a hit according to $alist_B$ and a miss according to $alist_A$. However, to accurately determine this difference between $alist_A$ and $alist_B$ is very complex and the algorithm is based on a more simple assumption. To begin with, it is assumed that all accesses in $alist_B$ do not exist in $alist_A$. The worst-case future access pattern will then consist of the accesses in $alist_B$ and Δ_{alist} will be the number of accesses in $alist_B$. This is, however, overly pessimistic. To get a useful algorithm, some additional operations are used. First, each access list is cleaned by removing all accesses that refer to a memory block that is later in the list referred to (superseded accesses). It is only the last access that occurred to a memory block that is used when determining hits or misses. Second, if the last element (or elements) in both lists are identical, i.e., they refer to the same memory block and belong to the same sequence, then they can be removed since these accesses can never cause a difference in the future.

It is worth noting that the algorithm presented in Figure 8.5 is quite pessimistic. In the next section, we evaluate this pessimism experimentally.

8.3 Experimental results

To evaluate the accuracy and time complexity of the presented algorithms, the improved conflict analysis has been integrated into the symbolic execution tool used in Chapter 5. Then, memory accesses from two different subroutines, *matmult* and *compress*, have been analyzed.

8.3.1 Experimental setup

The WCET tool used is based on the PowerPC architecture. In this evaluation, the pipeline and instruction cache analysis were turned off to focus the evaluation to the conflict analysis. Thus, the estimated WCET is simply set to:

$$\text{estimated } wcet = \max_{\mathbf{c}\in\mathbf{C}}(miss\_table(\mathbf{c}))P_{DC} + pen$$

For simplicity, we set the data cache miss penalty to $P_{DC} = 1$ to make the execution time equal to the number of data cache misses. The upper bound $\Delta WCET$ becomes:

$$\begin{aligned}\Delta WCET(t_A, t_B) &= \Delta_{conflict}(CA_A, CA_B) + \\ &\quad pen_A - pen_B\end{aligned}$$

The data cache parameters were set to 2048 bytes with a block size to 16 bytes. All pointers to objects were kept aligned with cache block boundaries.

The improved algorithm described in Section 8.2.4 was used. However, an extra optimization were added. By exploiting symmetry, it is possible to set the offset of one sequence to a fixed value and remove this sequence from the case variables. So if each offset of a sequence gets split into K cases, the total number of cases to examine will be K^{s-1} where s is the number of sequences.

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

	matmult	*compress*
	Multiplies two 10x10 matrices	Compresses 50 bytes of data
Identified sequences	4	3
Multiple paths	no	yes
Path merges	0	3965
Executed instructions	47994	46170
Data accesses made	8207	8005

Table 8.3: Benchmark properties.

Table 8.3 shows some properties of the benchmarks used. For *compress*, the instruction count and data access count is along the worst-case path found.

8.3.2 Data structure identification

The subroutine *matmult* has three incoming parameters being pointers to the three matrices R, A, and B, where R is the result matrix, and A and B are the source matrices for the multiplication. These matrices were identified as predictable data structures but with an unknown placement (see Section 8.2.1). Also, the stack area is treated as a predictable data structure but with an unknown placement since the initial stack pointer is considered to be unknown input data. In total, we identified 4 different memory access sequences.

In *compress*, the incoming parameters are pointers to the source and destination text buffers. The source text buffer was identified as predictable but with an unknown placement. However, the destination buffer was identified as unpredictable. This buffer must therefore be allocated to a non-cacheable memory area. Furthermore, *compress* was found to access the stack area as well as many global variables. Again, the stack area was treated as a predictable data structure with an unknown placement. The global accesses target both predictable and unpredictable data structures. The unpredictable structures were allocated to a non-cacheable memory area, while the predictable data structures were assumed to have an unknown placement since they could get allocated to a different place if the program is relinked. Also, the global variables were treated as one

compound object, i.e., the linker is assumed to allocate these variables in the same order regardless of the final placement. In total, we identified 3 different memory access sequences targeting the source text buffer, the stack area, and the global variables.

8.3.3 Metrics

The worst-case number of data cache misses has been estimated for different data cache associativity (direct-mapped, 2-way, and 4-way), and using different number of cases (see Section 8.2.4), splitting each offset of a sequence into $K = 1$ case (basic analysis) or $K = 2, 4, 8, 16,$ or 32 cases, giving a total number of cases of K^{s-1} where $s = 4$ for *matmult* and $s = 3$ for *compress*. The observed WCET has been included as a reference value. Ideally, we would have liked to include the actual WCET but this requires an exhaustive analysis that was too expensive to perform. Instead, we used guided testing to find the WCET. This was done by doing a traditional cache analysis for each tested case of initial pointer values. By using the result from the most accurate conflict analysis, we could speed up the testing by focusing on promising ranges of input values.

As a comparison, two more traditional data cache analysis methods have been used, *cache nothing* and *cache a single sequence*. The *cache nothing* method is simply to assume that all data accesses cause a cache miss or to turn off data caching. The *cache a single sequence* method is to only let one sequence be cached. Then, we have used a traditional data cache analysis using an arbitrary placement of the target data structure.

8.3.4 Conflict analysis results

Tables 8.4 and 8.5 show the results from the WCET analysis of the *matmult* and the *compress* subroutine. The observed worst-case tells us that when using the direct-mapped or the 2-way set associative cache, misses occur due to conflicts between accesses in the different sequences. The 4-way associative cache eliminates all conflict misses and only cold (compulsory) misses are left.

The basic analysis ($K = 1$) over-estimates the worst case for both the direct-mapped and the 2-way cache by a factor ranging from 1.16 (*compress*, direct-mapped) to 10.38 (*matmult*, 2-way). However, for the

Matmult		Total misses	Ratio
Direct-mapped	K=1	4204	3.36
	K=2	3828	3.06
	K=4	2963	2.37
	K=8	2143	1.71
	K=16	1567	1.25
	K=32	1252	1.00
	Observed worst	1252	
2-way set assoc.	K=1	4029	10.38
	K=2	2566	6.61
	K=4	1299	3.35
	K=8	730	1.88
	K=16	480	1.24
	K=32	389	1.00
	Observed worst	388	
4-way set assoc.	K=1	78	1.00
	Observed worst	78	
	Cache nothing	8207	
	Cache stack only	4103	

Table 8.4: Conflict analysis results for Matmult using improved algorithm with K cases.

4-way cache it finds the exact number of misses. To get tighter estimates for the direct-mapped and the 2-way cache, we need to split the analysis into cases. The algorithm managed to reach within 1 % of the exact worst-case number of misses when using $K = 32$ for *matmult* and $K = 8$ for *compress*. This is interesting since an exhaustive analysis corresponds to $K = 128$ for the direct-mapped cache. Thus, the exact estimate was found with considerable less effort than what an exhaustive analysis would require.

For *compress*, the result from the basic analysis is considerably lower for the 2-way cache when compared with the direct-mapped cache. This is expected since the 2-way cache, compared to the direct-mapped cache, allows the maximum age of a block to be higher before it gets replaced, leading to fewer cache misses. Somewhat surprising, the same is not true for *matmult*. In *matmult*, the basic analysis gives almost the same

Compress		Cache misses	Ratio	Merge penalty	Not cached	Total misses
Direct-mapped	K=1	3311	1.16	7	3001	6319
	K=2	3260	1.14	30	3001	6291
	K=4	3143	1.10	45	3001	6189
	K=8	2861	1.00	85	3001	5947
	K=16	2859	1.00	86	3001	5946
	K=32	2859	1.00	86	3001	5946
	Observed worst	2852		0	3001	5853
2-way set assoc.	K=1	563	3.61	54	3001	3618
	K=2	325	2.08	73	3001	3399
	K=4	189	1.21	86	3001	3276
	K=8	156	1.00	86	3001	3243
	K=16	156	1.00	86	3001	3243
	K=32	156	1.00	86	3001	3243
	Observed worst	156		0	3001	3157
4-way set assoc.	K=1	19	1.00	86	3001	3106
	Observed worst	19		0	3001	3020
	Cache nothing			0	8005	8005
	Cache stack only	6		0	5182	5188

Table 8.5: Conflict analysis results for Compress using improved algorithm with K cases.

result for the direct-mapped and the 2-way cache. The reason for this can be found by studying the memory access pattern in *matmult*. The typical data structure access pattern is: $A, B, R, stack, A, B, R, stack, \ldots$. This means that the possible maximum age of a previous access to the same data structure is 4. Thus, for *matmult*, the associativity of the cache must be greater than 3 to avoid counting many accesses as possible conflict misses.

In both subroutines, stack accesses are the most common type of access and have therefore been chosen as the access type to cache for the *cache single sequence* method. The results from this method reveals an interesting fact. For *compress* and the direct-mapped cache, it is better to only cache stack accesses than to cache all sequences. By caching only stack accesses, the number of conflict misses is reduced more than the increase of the number of misses from the data that is not cached.

	Matmult		Compress	
K	Total # cases	Analysis time [sec]	Total # cases	Analysis time [sec]
1	1	1	1	15
2	8	1	4	15
4	64	2	16	15
8	512	4	64	16
16	4096	25	256	20
32	32768	191	1024	37
Traditional		0.6		3.5

Table 8.6: WCET analysis times

Furthermore, for *matmult* and the direct-mapped cache, the *cache stack* method performs better than the basic analysis does, showing the need to split the analysis into cases.

8.3.5 Merge penalty

Table 8.5 also shows that for *compress*, a merge penalty was added for some merges to guarantee a safe estimate of the total worst-case number of misses. This means that differences in the list of accesses or the *miss_table* (see Section 8.2.5) was large enough to cause a penalty to be added. In our case, the difference is mainly due to the list of accesses and the use of the algorithm in Figure 8.5. This algorithm is quite simple and pessimistic. Nevertheless, the accuracy of the algorithm is quite sufficient for our study, since the added penalties are quite small.

Another explanation of the cause of the penalty is our use of a rather simple timing model. In this study, the execution time of a path is defined as being the number of data cache misses. If we would make the timing model more accurate and add the pipeline and instruction cache contributions to the execution time, the penalty would probably be reduced or eliminated since the execution time differences between paths to be merged would increase and this would in turn reduce the need for any added penalties.

8.3.6 Time complexity

An important question is how many cases the analysis can be split into without requiring too much analysis effort. The answer can be found in Table 8.6 which shows the analysis times for different number of cases and for the two benchmarks. The table also includes the times for doing a traditional data cache analysis (no conflict analysis).

Compared to a traditional analysis, the analysis time needed for a basic conflict analysis ($K = 1$) is a factor of 2 and 4 for *matmult* and *compress*, respectively. When splitting the analysis into cases, the extra analysis time needed is proportional to the number of total cases. As can be seen in the table, for small number of cases the extra analysis time is not noticeable. Therefore, a small number of total cases (for example 64) can always be used without slowing down the algorithm noticeably. Thus, for *compress*, the worst case is found with hardly no extra time needed. This is not true for *matmult* where we must spend a lot of time if we really want to find the worst case.

8.4 Related Work

Previously published data cache analysis methods [FW98, KMH96, LMW96, WMH+97, WE00] cannot analyze the interaction between different sequences of memory accesses. However, it is interesting to see to what extent they still can be used.

Previous methods are only able to handle a single access sequence. In order to handle multiple sequences, the problem must first be reduced in some way. There are several alternatives available and we have already used two methods in Section 8.3, the *cache nothing* and the *cache a single sequence* methods, that are both applicable. These two methods reduce the problem into analyzing at most a single sequence, which can be handled by all previously presented data cache analysis methods.

If the memory system lacks the possibility of not caching some parts of the memory, we will be forced to cache all data. In this case, the worst-case estimates will always become worse. We must count accesses to non-cached data as misses and if this data also becomes cached we must account for the possible additional conflict misses that may occur.

In this chapter, we have assumed a traditional cache architecture. However, there are several other cache architectures that could make the

problem easier or maybe remove the need for any detailed conflict analysis. For example, hardware cache partitioning [Kir89] could maybe be used to avoid conflicts by caching each data structure into a separate partition. Another way to avoid conflicts is to use a cache where the placement of data is software managed [Jac99]. However, these methods introduce the need to manage the partitions or placement of data. Traditional caches may be easier to use.

Even with traditional data caches it is possible to apply different software techniques to avoid conflict misses. For example, software cache partitioning [Mue95] or other conflict avoidance techniques [CKJA98] could be used. However, these techniques often require that a complete program is analyzed and produces a fixed placement of data. The algorithm presented in this paper focuses only on a single subroutine. It is not clear how the different techniques can be combined.

Chapter 9

Concluding Remarks

In the previous chapters, we have mainly tried to bring forward the strengths of the symbolic execution approach presented in this thesis. By doing an integrated path and timing analysis for all iterations of all loops, it can accurately estimate the WCET of a program. However, to make the method more practical to use a number of weaknesses should be remedied:

- **Time complexity** The greatest concern is probably the time complexity. To be more flexible, it should be possible to choose whether to do an accurate and slow analysis or a less accurate but fast analysis. Other methods use several techniques that could be borrowed. For example, to make the analysis faster it is maybe possible to adopt a branch-and-bound approach similar to the one used by Altenbernd [Alt96] to prune paths before they are simulated. However, this would influence the path analysis and the possible gain is not easy to predict.

  A more fruitful approach, is probably to include a possibility to avoid analyzing all the iterations of a loop by adding a data-flow analysis. The analysis done by the symbolic execution method can probably be defined as a traced based abstract interpretation [Sch98]. By adding a fixed-point algorithm, it should be possible to use a data-flow analysis when speed is more important than accuracy. However, this would destroy the information on loop bounds and the method would need to rely on manual loop bound annotations or be complemented by some other fast loop bound

  analysis [HSR+00]. An interesting approach, with moderate complexity, that might be used is the approximate loop analysis presented by Lim et al. [LBJ+94]. Ideally, we would like an approach that scales smoothly from full analysis of all iterations to a data-flow analysis using a fixed-point algorithm.

- **Modularity** Another way to achieve a faster analysis is to make a modular analysis. For example, by first analyzing a single subroutine in isolation and then reusing this result whenever this function is called reduces the complexity of the analysis. Further research is needed to find out how this can be done best in order to reduce the overestimation resulting from splitting the analysis.

- **Annotations** Clearly, the possibility to use manual annotations or path information from other kinds of analyses needs to be improved. The hybrid WCET estimation method presented by Wolf and Ernst [WE00] combines simulation of single feasible paths with the constraint solving approach. A similar technique could be adopted where the symbolic execution method is used to analyze all parts of the program where special constraints or annotations are not needed.

The goal of the research behind this thesis, as stated in Chapter 1, was to develop WCET analysis methods that could handle systems where the execution history can influence the timing of instructions. This covers arbitrarily complex systems. As we have seen in Chapter 6, the symbolic execution approach presented in this thesis have problems with hardware mechanisms having an unbounded timing effect. So, the approach cannot efficiently handle systems using such mechanisms. An interesting question is if it is possible to improve the analysis?

The timing analysis could possibly be made more accurate by incorporating knowledge about the whole program when doing the merge operation. The current analysis is a forward analysis, i.e., during the analysis no information about the future instructions in the program is available. For example, if we have knowledge of future memory accesses when calculating the upper bound Δ_{IC}, we can exclude many possible worst-case access patterns and reduce the upper bound. However, the gain from this can be limited. It will only improve the analysis of programs containing paths of similar length. It will not solve the problem

with unbounded timing effects.

The unbounded timing effects have different implications on the timing analysis depending on the application environment. For a strictly non-preemptive system (see Section 6.6) where all events are known and where the hardware allows control of the timing state in the system, the timing merge operation is no longer needed and the timing behaviour can be simulated using an arbitrarily accurate timing model. In this context, the timing analysis approach presented in this thesis probably represents the best analysis one can make.

In a preemptive system, pessimistic models must be used to handle all unbounded timing effects. However, this is a limitation that affects all timing analysis methods. Yet, it is possible that new kinds of analyses can improve the situation. For example, by analyzing all instruction really present in a program, it may be possible to construct a less pessimistic timing model valid only for that particular program. This kind of analysis would be fruitful for all timing analysis methods. For example, if it is possible to prove that no domino anomalies (see Chapter 6) can arise for a certain program, this can maybe be used to construct a less pessimistic model or to bound the possible timing effect so that a detailed timing model can be used for that particular program.

A big problem with the reliability of WCET estimation is the validity of the timing model when compared to the real hardware [Eng02]. To get an exact (or pessimistic) correspondence between the hardware and the timing model for a more complex system, it is probably necessary to formally try to relate the hardware description with the timing model. This only works if one have access to a formal specification of the hardware and if methods can be developed to relate these specifications to the timing model. For many commercial microprocessors where the exact implementation is unknown, it is still unclear if a reliable WCET estimation can be done. It can be more reliable to use measurement techniques or statistical modeling instead [CBG00, LHT00, PF99, BE00].

Bibliography

[AJ02] Björn Andersson and Jan Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'02)*, page To appear, April 2002. [100]

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002. [11]

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, June 1996. [6, 22, 23, 25, 149]

[AM00] Lars Albertsson and Peter S Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'00)*, pages 191–198, 2000. [11]

[AS92] P. Altenbernd and J. Strathaus. False path problem: Analyzing hierarchical designs of sequential circuits. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems*, pages 6–11, 1992. [25]

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986. [55, 68]

[BB00] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, 2000. [25]

[BE00] A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *Proceedings 12th EUROMICRO conference on Real-time Systems*, pages 89–96, 2000. [151]

[BN94] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994. [109, 121]

[Cag] Andrew Cagney. PSIM, a PowerPC simulator. http://sources.redhat.com/psim/. [59]

[CBG00] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, (LCTES'00), LNCS 1985*, pages 178–198, 2000. [151]

[CBW94] Roderick Chapman, Alan Burns, and Andy Wellings. Integrated program proof and worst-case timing analysis of SPARK ada. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages K1–K11, June 1994. [6, 22, 24, 25, 25]

[CKJA98] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceeding of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139–149, October 1998. [148]

[CP00] A Colin and I Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, May 2000. [57]

[CP01] A Colin and I Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, pages 37–44, June 2001. [7, 8, 57]

[EE00] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 163–174, December 2000. [55]

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, pages 1298–1307, August 1997. [6, 21, 22, 23, 25]

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, April 2002. [55, 151]

[EY97] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of International Conference on Computer-Aided Design (ICCAD'97)*, pages 598–604, 1997. [6, 22, 24, 25]

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT 2001), LNCS 2211*, pages 469–485. Springer, October 2001. [6, 7, 8, 22, 24, 25, 56, 86, 97, 98]

[FW98] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, June 1998. [109, 111, 115, 120, 121, 124, 147]

[FW99] Christian Ferdinand and Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17((2/3)), 1999. [55, 56, 57, 86, 97]

[GL02] Gustavo Gómez and Yanhong A Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 75–86, January 2002. [6, 22, 23, 25, 25]

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969. [100]

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000. [20, 23]

[HFL01] I. Hayes, C. Fidge, and K. Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings Software*, 148(6):175–186, December 2001. [25]

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2ed.* Morgan Kaufmann, 1996. [27]

[HPRA02] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *IEEE Computer*, 35(2):40–49, February 2002. [11]

[HSR+00] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):121–148, May 2000. [6, 22, 24, 25, 150]

[HW99] Christopher A. Healy and David B. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 79–88, June 1999. [6, 22, 24, 25, 57]

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995. [7, 8, 57, 86, 97, 97, 98, 109]

[IBM] IBM. PowerPC 403 GA user's manual. http://www.chips.ibm.com. [75, 113]

[Jac99] Bruce Jacob. Hardware/software architectures for real-time caching. Presented at CASES'99: Workshop on Compiler and Architecture Support for Embedded Systems, Washington DC, October 1999. [148]

[Kir89] David B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 229–237, December 1989. [148]

[KMH96] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996. [7, 8, 38, 56, 86, 97, 109, 111, 120, 120, 121, 122, 124, 147]

[KW98] Apostolos A. Kountouris and Christophe Wolinski. False path analysis based on a hierarchical control representation. In *Proceedings of the 11th International Symposium on System Synthesis*, pages 55–59, 1998. [25]

[LBJ+94] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 97–108, December 1994. [7, 8, 56, 86, 97, 98, 109, 150]

[LHKM98] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 334–345, December 1998. [98]

[LHT00] Markus Lindgren, Hans Hansson, and Henrik Thane. Using measurements to derive the worst-case execution time. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications, (RTCSA'00)*, pages 15–22, December 2000. [151]

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995. [7, 8, 22, 55, 86, 97, 109]

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996. [7, 8, 55, 55, 86, 97, 109, 111, 120, 121, 124, 147]

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, LNCS 1474*, pages 1–15. Springer, June 1998. [8]

[LS99a] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999. [8]

[LS99b] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262. IEEE, December 1999. [8]

[LS99c] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 12–21, December 1999. [8, 87]

[Lun02] Thomas Lundqvist. Data cache timing analysis with unknown data placement. Technical Report 02-11, Departement of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, February 2002. [9]

[MDG+98] P Magnusson, F Dahlgren, H. Grahn, M. Karlsson, F. Larsson, A. Moestedt, J. Nilsson, P Stenström, and B. Werner. Simics/sun4m: A virtual workstation. In *Proceedings of USENIX98*, pages 119–130, 1998. [11]

[Mue95] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995. [148]

[Mue97] Frank Mueller. Timing predictions for multi-level caches. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1997. [109]

[Mue00] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000. [7, 8, 57]

[NP93] V. Nirkhe and W. Pugh. A partial evaluator for the maruti hard real-time system. *Real-Time Systems*, 5(1):13–30, 1993. [6, 22, 23]

[OS97] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 47–55, June 1997. [7, 8, 55, 55, 55, 86, 97, 109, 111]

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993. [22]

[PB01] Peter Puschner and Guillem Bernat. Wcet analysis of reusable portable code. In *Proceedings of the 13th International Euromicro Conference on Real-Time Systems (ECRTS'01)*, pages 45–52, June 2001. [25]

[PF99] S. Petters and G. Färber. Making worst-case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications, (RTCSA'99)*, pages 442–449, December 1999. [151]

[PK89] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989. [6, 7]

[PS97] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997. [7, 8, 22, 55]

[SA00] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000. [7, 8, 23, 57]

[Sch98] David A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of the ACM Symposium on Principles of Programming Languages, (POPL'98)*, pages 38–48, January 1998. [149]

[Sch99] Jörn Schneider. Personal communication, November 1999. [87]

[Sch00] Jörn Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 195–204, November 2000. [107]

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, pages 35–44. ACM SIGPLAN Notices, volume 34, May 1999. acceleration effect. [87, 95]

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989. [7, 56]

[TF98] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, December 1998. [97, 109]

[VHMW01] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems (LCTES'01)*, pages 88–93, June 2001. [25]

[WE00] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, November 2000. [7, 8, 56, 56, 86, 97, 109, 120, 120, 124, 147, 150]

[WMH+97] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3nd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997. [57, 86, 97, 109, 111, 120, 121, 124, 147]

[WMH+99] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, November 1999. [7, 8]

The numbers after an entry list the pages that have a reference to that entry.