Instruction Cache Memory Issues in Real-Time Systems

Filip Sebek
Computer Architecture Lab
Department of Computer Science and Engineering

Technology licentiate thesis
2002-10
ISBN 97-88834-38-7

(14th October 2002)

Instruction Cache Memory Issues in Real-Time Systems

Filip Sebek

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

MÄLARDALEN UNIVERSITY

Abstract

Cache memories can contribute to significant performance advantages due to the gap between CPU and memory speed. They have traditionally been thought of as contributors to unpredictability because the user can not be sure of exactly how much time will elapse while a memory-operation is performed. In a real-time system, the cache memory may contribute to a missed deadline by actually making the system slower, but this is rare. To avoid this problem, the developers of real-time systems have run the program in the old-fashioned way; with disabled cache — just to be safe. Turning the cache off, however, will also make other features like instruction pipelining less beneficial so the new processors will not give the performance speedup as they were meant to give.

The first methods to determine the boundaries of the execution time in computer systems with cache memories were presented in the late eighties — twenty years after the first cache memories were designed. Today, fifteen years later, further methods have been developed to determine the execution time with cache memories . . . that were state-of-the-art fifteen years ago.

This thesis presents a method of generating worst-case execution time scenarios and measure the execution time during those. Several important properties can be measured. These include cache-related pre-emption delay, miss-ratio levels of software, and instruction cache miss-ratio threshold levels for increased system performance. Besides the dynamic measurement method, a statical procedure to determine the maximum instruction cache miss-ratio level is presented.

Experimental results from this research show that the indirect cache cost of a pre-emption is very high — more than three times the execution cost of the context-switch functions themselves. Another result shows that the tested computer system without caching will not cause a missed deadline if the instruction cache is enabled.

ISBN 97-88834-38-7
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

This document was written in Emacs 20.3.1, typesetted in LaTeX, spell and grammar checked in Microsoft Word 2000 and compiled with MikTeX 2.2. Figures were created by `gpic` from the GNU organization and `dot` from AT&T labs.

To Linda, Zacharina, and Xerxes

Preface

It all began that day in 1981 when my father came home with the book “Basic Basic” by Dag Toijer which he had borrowed from the library. I read the book several times and started to write programs — with a paper and pen since a computer was far too expensive to own. Four years later I bought my first computer — a Commodore 64. I will never forget that day in Hanover, walking proudly with the box containing my very own computer.

Many books and computers later, in 1998, I had the opportunity to become a Ph.D. student while working as a teacher at Mälardalen University. The Department of Computer Engineering performed research in real-time systems, and since my main interest was in computer architecture, I found this combination of topics to be natural but also very exciting. It has been a challenge to struggle with unreadable papers, bugs in my own software, and glitching hardware not to mention all the other duties I have as a teacher and being a part of my own family. Several times I have been close to giving it all up, but today I’m finally there.

The work presented in this thesis would probably have been impossible if I had not had such encouraging and stimulating people around me. My supervisors Lennart Lindh, Björn Lisper, Hans Hansson and Jan Gustafsson have always been available to me and giving me fantastic support, feedback, input and inspiration to my immature ideas and questions. Per Holmberg’s great feedback has really improved the contents and structure of this thesis. Thank you! A very special thanks goes to Mohammed El Shobaki who has always criticized my work in the most creative way, helped me with problems and been such a good friend. Even if the spell and grammar check is quite good in MS Word, this thesis would doubtfully be readable without Roxanne Fitzgerald’s thorough proofreading. Other thanks go to Johan Stärner, Leif Enblom, Peter Nygren, Gustaf Naeser, Jeroen Heimans, Raimo Haukilahti, Xavier Vera,

Jonas Neander, Mikael Åkerholm, Joakim Adomat, and Mogens Jensen for all their valuable help and input. Other people in the department who have been of importance to me are Christer Norström, Gunilla Eken, and Monica Wasell. They have solved virtually all of the problems that arose when my research conflicted with all my other work at IDt. Friends such as these don't grow on trees and I'm lucky to have them.

The greatest debt I owe is to my family; Linda, Xerxes, Zacharina, and my parents. They have all been almost fanatically supportive and enthusiastic in making it possible to finish my work.

Västerås, the warm days of August 2002.

Contents

Chapter 1

Introduction

1.1 Cache memories in computer systems

The traditional computer is usually designed with a *central processing unit* (CPU), a primary memory to store program and data, and input/output-devices (I/O) to communicate with the environment. The silicon based electronic components have doubled in capacity every 18 months during the last 30 years. This improved capacity has been used to creat larger primary memories and faster CPUs (or actually physically smaller so that the chip is able to run at higher clock frequencies due to shorter signal paths). The problem with this trend is that the memory is unable to keep pace with the CPU and feed it with

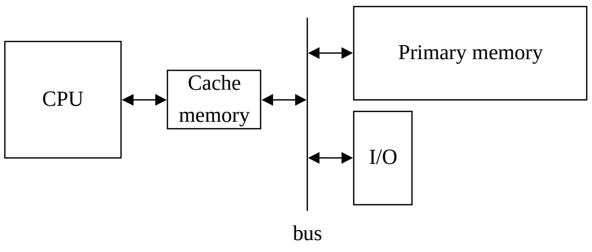

Figure 1.1: The cache memory in a computer system.

new instructions. One solution for bridging the widening performance gap between CPU and memory, is to use a small but fast *cache memory* (Figure 1.1).

Only a fraction of the instructions and data located in the primary memory can be held in the small memory. This means that if a requested memory block doesn't reside in the small memory it must be fetched from the slower large memory. The more a memory block is reused before it is swapped out, the higher performance can be achieved.

The difference between a *buffer* and a cache memory is that a buffer is controlled by the program(mer) in the same way as a CPU register, and a cache memory is dependent on a software property called *locality* which can be either temporal or spatial[Smi82, AHH89].

- *Temporal locality* (also called *locality in time*) concerns time. If a program is accessing an address, the chances are higher that this same address will be reused in the near future, as opposed to some arbitrary address.

- *Spatial locality* (also called *locality in space*) states that items that are close to each other in address space tend to be referred to close in time too.

The bulleted statements builds on the fact that instructions are formed in sequences and loops, and that data is often allocated as stacks, strings, vectors and matrices. To take advantage of the greater bandwidth and longer latency which characterizes primary memory, and keeping the spatial behavior of programs in mind: it makes sense to load more data at once, than just that which is requested from the underlying hierarchic memory

This chunk of data is called *cache line* or *cache block*, and is the smallest piece of data handled by the cache memory. The CPU requests regularly a data or an instruction, and if it is in the cache it is called a *hit*. Otherwise it is a *miss*, resulting in a primary memory fetch penalty. A high hit-ratio constitutes high performance. With a low degree of locality of the program or a program that doesn't care about cache memory behavior one can expect low performance and a longer response time. This means that the performance and the contents of the cache memory are controlled indirectly by the design of the executing program.

Today's modern CPUs have integrated instruction and data caches on the same chip as the CPU core and second and third levels of larger caches bridge the gap between the first level cache and the primary memory. Besides this hierarchy of caches, features such as *non-blocking caches*, *victim caches*, *write buffers* and *critical word first* makes the system more complex, but yields even higher performance.

1.2 Real-time

1.2.1 Overview

In the last few decades, the computer has replaced automatic control and traditional control systems in industrial applications. It has succeeded PLCs and manual gears due to its flexibility, accuracy, and speed in numerous areas. Examples of the use of automatic control systems, or real-time systems (RTS) as they are called nowadays, can be in a nuclear power plant, an air bag in a car, the power control in a microwave oven, a robot which paints automobiles or in a telephone switching device. In all these systems time and timing is crucial. The response time cannot be exceeded or the system will fail.

Controlling a process with a computer (real-time) system is achieved by the following steps (in order)

1. Observe the process – *sample* values from a probe
2. Decide what to do – execute instructions to calculate new positions etc.
3. Actuate – control the process by giving a signal to a controlling device (motor, relay etc.)

This approach raises two new questions; how often must a sample be taken and how fast must each sample be analyzed by the computer to give a correct control signal? If the sample isn't performed often enough, the system will have a very rough and "jumpy" view of the world. A higher sample rate will give a smoother view, but also more data to handle. The sampling process should therefore not be performed more often than necessary. The second issue regarding the computation (and possibly even physical) delay is perhaps even more important. It is possible to achieve a fast response time with a small and simple calculation, but a more sophisticated algorithm with more precision takes longer to calculate. If the response time is too long the system will not work as intended and possibly cause errors and injuries. In this case, the system is controlling the process as it was, and not as it is, in "real time".

The answer to both questions is that the environment in which the system interacts must be specified and from this specification, one can get answers such as s sample rate of no less than 500Hz or worst-case response times of no more than 10 milliseconds.

A task must always be finished before a *deadline* but in many applications it might also be impermissible to complete the task before a certain point in time. In a *hard* RTS all tasks must be completed in this window of time but in *soft* RTS a specified percentage may fail to do this. Hard RTS are typically those which control something that must not fail because failure will cause process malfunction or damage — for instance a painting robot must stop a movement in time or the surface will be scratched. Soft RTS are those that can tolerate misses to some extent and will regulate the process with decreased quality of the result — for instance a telephone switch station. The specifications of the system determine if it is a hard or soft real-time system.

1.2.2 Scheduling

The simplest form of a controller of a real-time system, is a single program running in a computer with no other tasks to perform. More complex system controllers are however easier to construct with several tasks (processes, threads) running simultaneously in a time-sharing environment in which the system resources are more efficiently utilized. An ordinary operating system gives every task an equal slice of the time and runs the jobs in a round-robin order. An operating system that is specialized for real time systems can utilize performance even more by prioritizing tasks, permitting pre-emption (interrupting low-priority jobs with high-priority jobs before resuming), using more advanced scheduling algorithms, etc.

Scheduling can be performed at run-time (dynamic) or in advance (static). Theoretically, only a few applications can be designed to use a static schedule which can utilize system performance up to 100%. It is easy to prove that a schedule prepared in advance will meet time constraints. Systems that are event-driven can usually utilize system resources well if they are scheduled at run-time; if certain events occur only sporadically a static schedule must have allocated enough time for each task period where all cyclic tasks has finished their execution. An example of a static scheduling algorithm is “rate monotonic”[LL73]’. The principle behind rate monotonic is to assign the highest priority to the task with the shortest time period and the lowest priority to the tasks with the longest time period. This scheduling algorithm has been proven to work for a CPU-utilization of maximum 69% when the number of tasks is infinite.

“Earliest deadline” is a dynamic scheduling algorithm which can dynamically change the priority of the tasks during run-time. The key concept is to assign the highest priority to the task with shortest time until deadline. This leads to the possibility of higher system utilization than is possible with, for instance, the rate monotonic algorithm. The main disadvantage of “earliest deadline” is that if a task misses its deadline, the execution order becomes undefined.

1.2.3 Execution time analysis

Real-time systems rely on correct timing. All scheduling needs timing data and in this instance, “real-time” literally means *real time*. To schedule a task-set in a way that no task ever will miss deadline, each task must be provided with the *execution time* in, for instance, milliseconds. This execution time can for instance be calculated statically, measured when the computer runs the execution path that yields the longest execution time or just ”known” in somé way. In many cases the *worst-case execution time* is used to be safe since the execution and the relationship between the tasks may influence on the execution and thereby the execution time also.

To determine the execution time of a program sequence is seldom as easy as taking a stopwatch and measuring the time it takes to execute a program section. The response time of a program depends not only on the critical code sequence to be executed (whatever that is) but also on computer performance, workload, condition, state, etc. and therefore an analysis must be performed.

The execution time of a program is dependant upon the following factors [Gus00]:

The input data — The value of the data may determine the execution path and the number of iterations in a loop. The timing of the data is also important; data may come irregularly or in bursts.

The behavior of the program (as defined by the program source code) — The design of the program (tasks and operating system) can be short, long, data intensive, interrupt-driven, polling, short or long time-slices, task layout on multiprocessors . . .

The compiler (the translation from source to machine code) — Execution time can be substantially affected by loop unrolling, use CPU of registers instead of main memory etc.

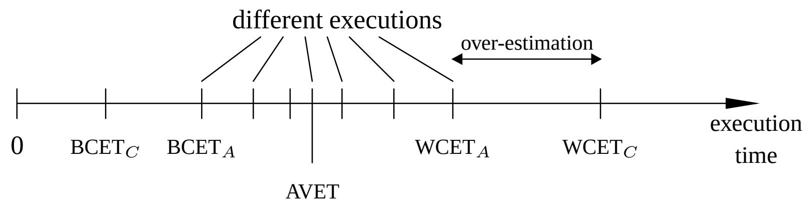

Figure 1.2: Basic execution time measures (from [Gus00])

The hardware — Number of processors, processor type, bus bandwidth, memory latency, cache memory and instruction pipelines determine how fast the machine instructions will be executed.

Although all of these factors affect the WCET, the best-case execution time (BCET) can be equally important (e.g. a too-early inflation of an air bag in a car can be fatal for the driver). Preferably, execution time analysis is performed statically, which means that it is *calculated* (index C) and overestimated from *actual* (index A) execution time since the analysis must take a very pessimistic approach, such as including non-feasible execution paths and maximal iterations of all loops. Figure 1.2 illustrates the basic execution time measures.

1.3 Cache memories and real-time

Calculating a tight WCET$_C$ in a system with cache memories is very tricky because the contents of the cache memory depend on the program's former execution path. On the other hand, the execution path depends (to some extent) on the cache contents!

A naive approach to calculating WCET$_C$ would be to handle all memory accesses as misses but that would result in a WCET$_C$ which would be more overestimated than a system without a cache (since the miss penalty time is longer than a simple regular primary memory access). In that case, it would be better to disable the cache as has been done in safety-critical hard real-time systems.

There are two categories of block swap-outs [AHH89];

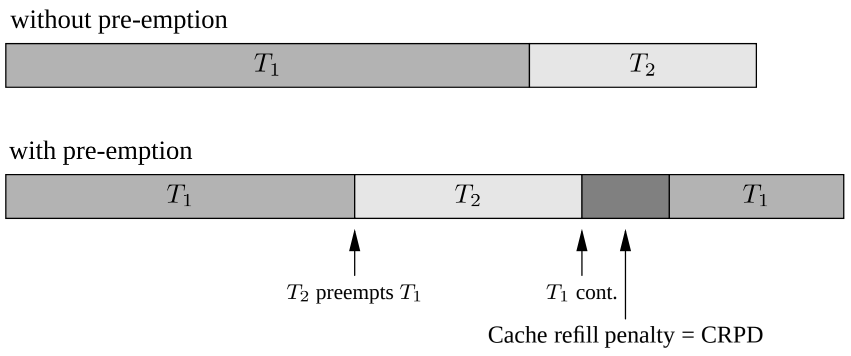

Figure 1.3: Cache-related pre-emption delay. A pre-empted task can be drained by cached data and suffer a refill penalty. Note that the penalty does not come right after resuming the pre-emption — it may come later or in pieces depending on the program's design.

- intrinsic (inter-task) behavior depends on the internal design and execution path of the task. Two functions or data areas in the task may compete for the same cache space and increasing the cache size and/or associativity can reduce the effects.
- extrinsic (intra-task) behavior depends on the environment and the inter-task behavior of the other tasks. At context-switch (pre-emption) the cache contents will be more or less displaced by the new running task. This performance loss is also called *cache-related pre-emption delay* (CRPD) or *cache refill penalty*. The CRPD, considered an *indirect cost*, is illustrated in Figure 1.3. In [BN94] the pre-emption's impact on WCET$_C$ is defined as:

  $$WCET'_C = WCET_C + 2\delta + \gamma \tag{1.1}$$

  , where $WCET'_C$ is the cache affected WCET$_C$, δ is the execution time for the operating system to make a context-switch (two are needed for a pre-emption), and γ symbolizes the maximum cache-related cost of a pre-emption.

Even if both kinds of swap-outs can be reduced, the real-time aspect remains as long as the hit-ratio is less than 100%. Some methods to show how the cache performance can be predicted will be presented in the next chapter.

Chapter 2

Related work and motivation

As mentioned in the introduction, cache memories are unpredictable components in a real-time environment since the execution time of each instruction becomes variable. There are several approaches to cope with this problem but none of them are perfect and all of them have certain drawbacks. This chapter will present a few methods to incorporate cache memories into real-time systems safely and then motivate an alternative method.

2.1 Related work

This section will give only a brief survey of some of all existing methods to give the reader a snapshot of the research area. The more interested reader can refer to [Seb01] in which the state-of-the-art cache memories and real-time are described.

2.1.1 Avoidance methods

There are several methods of avoiding or reducing the negative cache memory effects of real time systems. Here are some examples

RT designed processors . The MACS processor approach presented in [CS91] is to avoid the necessity for caches by using memory banks. The execution of tasks is performed on a single shared pipeline and all tasks are run at “task level parallelism” with instructions. Instead of making a context switch at regular millisecond intervals, each task in succession feeds a new instruction to the pipeline. In other words: if N tasks are running in the system, a new instruction will be executed in a specific task, every N clock ticks.

Another specially-designed processor (micro controller) with simple predictable behavior was the RTX2000 by Harris developed in the early 1990's.

Prefetching Since the real time operating system knows when a context-switch is to be performed, the cache could be prefetched with the necessary instructions and data [Stä98]. Like all prefetching, the tricky part is timing and the cache memory must also have a non-blocking feature.

Scheduling If a schedule allows pre-emption only at special points in the program[LLL$^+$98] the CRPD can be more easily determined. If pre-emption is impermissible, the CRPD will be eliminated and the extrinsic cache behavior can be determined with an intrinsic behavior analysis method. Limiting pre-emption leads to slower *response time* for low priority tasks.

Cache partitioning By reserving a private part of the cache memory for each task, extrinsic cache behavior can be eliminated. This can be performed either in hardware [Kir88, Kir89] or by linking tasks to memory addresses not interfering with each other in the cache — so called *software partitioning* [Wol93, Mue95]. The major drawback of partitioning is that each task will have access to only a fraction of the complete cache with a performance loss as a result. Data coherency must also be maintained since data structures can be duplicated in more than one partition.

2.1.1.1 Discussion

Avoidance is a safe and mature way of solving the problem with unpredictable hardware, but has some drawbacks. Reduced performance is maybe the most obvious, but this can be a reasonable price if no further analysis will be necessary. To put restrictions into a system might also be an educational problem since developers might have become accustomed to doing “everything they want” — new thinking can result in new type of errors. Other possible drawbacks with specially-designed processors, software or other components are the increased costs of developing and integrating these components into existing systems, and in some cases, higher system energy consumption.

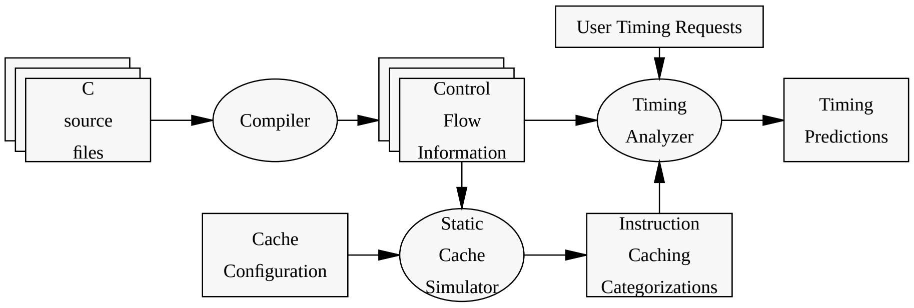

Figure 2.1: Overview of the flow to determine the bounds of instruction cache and pipeline performance.

2.1.2 Static analysis

Static analysis means that the software is examined without being executed. For instance, there are several approaches that are based on *integer linear programming* [LM95, OS97], *data flow analysis* [AMWH94, LML$^+$94], *graph coloring* [Raw93], *abstract interpretation* [AM95, FMWA99], and *reducing* [NR95, LS99b, SA97, TD00] among others.

To keep things brief, only two of these methods have been chosen to illustrate what static analysis is all about.

2.1.2.1 Example 1

This method was developed at Florida state university and is often referred as "static cache simulation" [AMWH94]. For an overview of the analysis flow see Figure 2.1.

By simulating a program in a static cache simulator, each cache block can be categorized as *always hit*, *always miss*, *first miss* or *first hit*. After the cache simulation the *timing analysis* is performed to determine WCET_C . The tool interactively asks the user to assign each loop the maximum amount of iterations that the compiler couldn't automatically determine. In the next step, the analyzer constructs a *timing analysis tree* and the worst-case cache performance is estimated for each loop in the tree. After this step, the user can request timing information about parts, functions or loops in the program. This method

time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
access event	-	4	5	6	7	0	0	1	2
action	-	m	h	h	m	m	h	m	m
cache block 0	0	**4**	4	4	4	**0**	**0**	0	0
1	5	5	**5**	5	5	5	5	**1**	1
2	6	6	6	**6**	6	6	6	6	**2**
3	3	3	3	3	**7**	7	7	7	7
useful blocks	{5,6}	{5,6}	{6}	{}	{}	{0}	{}	{}	{}

Figure 2.2: A memory access scenario from which useful cache blocks have been derived.

has been extended to handle set-associativity, data-caches [WMH+97], wrap-around-fill [WMH+99], and instruction pipelining [HAM+99]. This method has been criticized to have a too conservative view of the categorization model in the simulator[LBJ+95].

2.1.2.2 Example 2

The approach of Chang Gun Lee *et al* [LHS+96, LHM+97] to determine a tight CRPD is to identify and use only the *useful cache blocks* in the calculation. This analysis is performed in two steps;

1. **Per-task analysis**: Each task is statically analyzed to determine the pre-emption cost at each execution point by estimating the number of *useful* blocks in the cache. In the example illustrated in figure 2.2 the cache contains memory blocks number $\{0, 5, 6, 3\}$ during the time period t_0.

   The technique to determine the amount of useful cache blocks is based on data flow analysis with the control flow graph (CFG) of a program as input. One array stores the blocks that are reachable (*reaching memory blocks* – RMB) and another stores *live memory blocks* (LMB) at each execution point. The amount of useful blocks at each execution point can be determined from this material with an iterative method. The worst-case pre-emption scenario turns up when a task is pre-empted at the point with the largest amount of useful cache blocks since the refill penalty will be largest when the task resumes its execution.

2. **Pre-emption delay analysis**: A linear programming technique is used to

compute γ^1 by sum up the products of the number of task pre-emptions during a given time period, with the number of useful cache blocks at an execution point (taken from the pre-emption cost table).

2.1.2.3 Discussion

Both of these approaches and solutions for calculating WCET and CRPD are probably the best and most accurate today, but they are complicated and need much expertise and hands-on work. The methods are safe and applicable on hard real-time systems, but are not always able to model all the newest features of modern microprocessors such as trace caches, prefetching and read/write buffers. Data caches are still being handled very pessimistically since no method to the authors knowledge can handle dynamic memory. All those accesses are considered as a full penalty with a write-back and a cache line refill (also referred as "double miss")[KMH96, LS99a].

2.1.3 Simulation

By feeding extracted execution traces to a simulator it is possible to monitor internal states and measure execution time. The simulator permits changes of architecture parameters (cache size, band width etc) to make the analysis more flexible.

Mogul and Borg measured cache-related pre-emption delay as 10-400 microseconds by using trace driven simulation[MB91].

2.1.3.1 Discussion

One problem arises if the execution path (i.e. the trace) changes due to the timing of input data. Since the execution time depends on the execution path and cache contents, the conclusion is that the simulation must model the system from which the traces were derived. As will be described in the motivation (Section 2.2), modeling a system (for simulation) correctly is very difficult.

2.1.4 Real measurement

Real measurement is performed directly on the target system with software (SW), hardware (HW) or a hybrid combination of these (HW/SW).

[1] by Lee *et al* referred as $PC_i(R_i^k)$

To measure with SW or HW/SW monitoring methods, *probes* are inserted in the code. For example, probes can be implemented as *kernel probes* that are incorporated in the OS-code of task-switches, system functions, interrupt routines etc. *Inline probes* are placed directly in the application code and *probe tasks* which log the system state.

Solutions with software involved are able to give a high level of information but the price paid is that the measurement itself affects both the system (e.g. cache contents) and the execution path (e.g. execution time). If probes are removed after the measurement, a *probe effect* may occur, therefore, all probes should remain in the program. To make a system observable (and measurable) you must allocate enough resources and performance to the monitoring equipment even after the test is performed [Tha00].

Measurements with an oscilloscope on a bus will not affect the execution time, but many details may be hidden since most modern CPUs integrate cache memories and other components in the same capsule.

2.1.4.1 Example

In [PF99], Petters and Färber describe a measurement approach that begins with an automatically compiler generated Control Flow Graph (CFG). The CFG reflects the optimized code and a *reduced CFG* is produced after an analysis which cleans non-feasible paths and omits non WCET-paths. The reduced CFG is manually partitioned into measurement blocks to reduce complexity and hereby decrease the time of the measuring phase. The last step before the actual measurement is to insert probes into the measurement blocks to provide an external host with time-stamped identification-tags for post-monitoring.

This method has been implemented with a tightly coupled multi-processor system on a PCI-bus with MIPS4600 and dual Intel Pentium II/III CPU-high performance units (HPU). The system also contain a Real-Time Unit[2] , Configurable I/O Processor (CIOP) with FPGA and dual ported RAM. The system goes under the name REAR.

The testing is performed by manipulating the object code to select execution paths and a measurement is then performed. The next step is to evaluate the result, make a new manipulation, possibly avoid a code section, and start new measurements. This is performed *iteratively* and to reduce complexity and manual work, the measurement is performed from “good”, “safe”, low-complex points in the code at loops, function calls etc. However, the method needs much hands-on work and knowledge of how the software is constructed

[2]A dedicated Intel Pentium II CPU with a local SRAM which runs the operating system

to give safe and a fast WCET$_C$. It has also been criticized as being insufficiently safe for use in *hard* real-time systems. The attraction of the method is that very complex systems can be analyzed with a WCET$_C$ close to WCET$_A$.

2.1.4.2 Discussion

The best argument for using real measurement is that the execution time is actual and not over-estimated. The method is applicable on complex “state of the art” hardware that might be difficult, very time consuming, or even impossible to model with, for instance, a simulated or static method. The method has, however, been considered as unsafe in academia and by developers of safety critical real-time systems. Real measurement seems to be difficult to generalize in different kinds of systems, since the published methods (to the best of the author’s knowledge) need hands-on work and particular knowledge of the system and its micro-architecture to be performed correctly.

2.2 Motivation

At the beginning of this research I had a couple of questions;

- Are there simpler ways to analyze or measure cache properties than those already proposed in established research papers? (This question is answered in all the papers if those methods can be considered as “simple”)
- What kind of cache information can be tapped from a high performance computer system running at several hundreds of megahertz without interference? (answered in papers A and B)
- Can disabled instruction caches cause missed deadlines if they are enabled? (answered in paper D)
- How much is the cache-related pre-emption delay in a system in absolute and relative terms? (answered in paper B)

Much research has been performed to determine the (worst-case) execution time of a program. The execution time is necessary input for the scheduling algorithms. One problem is that most analysis methods and simulations models are simplified and not as complex as modern processors and computer systems, so that in many cases their usability is limited. Modern computer systems use complex microprocessors and system developers need real true values to schedule those systems.

The objective of this thesis is to show that a measured value is, in almost all cases, safer than one calculated since calculations need correct models. The model may be incorrect due to simplifications but also due to flaws and bugs in the system specifications and manuals [AKP01, Eng01]. Since a measured value isn't always reproducible due to the unknown initial system state before the measurement, the method is only safely usable in statistical analysis, i.e. in soft real-time systems.

Chapter 3

Results and contributions

This chapter will briefly present the results of the research performed. References for more detailed descriptions in the papers, conclude each section.

3.1 The measurement method

The method of measuring cache activities relies on a built-in processor *performance monitor* and a high-resolution *timer*. Today, the performance monitor is commonly built into the processor core as a tool which the programmer uses to optimize the program for the processor. Dedicated registers count specified events such as cache misses or instructions executed and this data can be used for profiling the running code. The timer is used to timestamp all events for post analysis. (Section 5.3, 6.3)

3.1.1 Generation of workbench code

The goal of this method is to measure the time of worst-case cache-related pre-emption scenarios. A common method to test a computer performance is to use standardized benchmarks to make systems comparable. However, no benchmark (e.g. "Rhealstone" [Kar90] or EEMBC) evaluates worst-case scenarios. Since the cache-related pre-emption delay is dependent on cache and code size, each test bench is unique for a system, and easiest to generate automatically with synthetic code.

3.1.1.1 Code with a specific size

It is easy to generate code of an exact specific size by inlining assembly instructions into a task, each instruction being a four-byte word. In this case the code only contained `r1=r1+0`. Load and store instructions were not used since only instruction caching was studied. Instruction pipelining is sensitive to instruction dependencies and the instruction mix, but since the impact of pipeline effects on execution time only is a fraction of cache issues, they have been ignored. (Section 6.3.4.1)

3.1.1.2 Code with a fix cache memory miss-ratio

To verify that the measured cache miss-ratio was correct, code with a known cache miss-ratio had to be generated. One way of achieving this, is to insert jump instructions in each cache block which will interrupt the spatial locality. A high miss-ratio can be achieved only by executing a fraction of the instructions in the block. If the task is larger than the cache memory, all blocks must be fetched from the primary memory and will prevent the utilization of temporal locality. The code only contains `r1=r1+0` and "branches (always) to the beginning of the next cache block" with the same argument as the size-specific task generation. (Section 6.3.3, 8.2)

3.1.2 Experimental measured results

The complex target system "SARA"[LKF99] has been used to obtain experimental results and to test the measurement method. The next paragraph will describe an overview of the system's features and special hardware components. Note that all the hardware features are not necessary for the generalized measurement method — they only make the method easier to use with some support.

SARA is based on the *microprocessor MPC750* from Motorola that is equipped with a split instruction and data cache at the first level, and a unified cache memory at the second level. The MPC750 also has an on-chip performance monitor that can monitor 48 different kind of events in four performance monitor counters[Mot97, Mot99]. The *Real Time Unit* (RTU)[FSLA95] is a high-performance and performance-predictable hardware-implementation of an operating system kernel that handles scheduling and other real-time operating system services. A special device designated *Multipurpose Application Monitor (MAMon)* [ES01, ES02], can tap the RTU non-intrusively on information regarding for instance context-switching, inter-process communication, task synchronization etc. The application may also write directly to special registers designated *software probes*. All the data collected in MAMon is sent through a parallel port to an external host for post-analysis.

Execution time, cache miss-ratio, context-switch time, threshold miss-ratio for an enabled/disabled instruction cache and maximum cache-related preemption delay (CRPD) have been measured and determined with a high degree of accuracy. (Paper B)

3.2 Determining the worst-case cache miss-ratio

The approach to finding the execution path with the highest cache miss-ratio, is to identify branches and target addresses in cache blocks to determine the miss-ratio of the spatial locality. In essence, it is the reverse process of generating code with a fixed cache miss-ratio, although an arbitrary code is much more complex than a synthetic workload with loops and conditions that must also be analyzed. The cache analysis must therefore be combined with an execution path analysis to calculate a miss-ratio level.

The execution path that yields the highest cache miss-ratio need not be the one with the longest execution time, but this path is the most energy consuming for the memory components. (Paper C)

3.3 Summary of papers

Four published papers have been included with this thesis, which explain the work performed in more detail than this chapter.

3.3.1 Paper A

This paper suggests two different methods of measuring the cache-related preemption delay. The first method is a complete hardware solution using JTAG to tap information from the CPU's performance monitor. The second method is a HW/SW hybrid method which stores no data on the target system, but sends it continuously to a database host through a parallel cable. This is preferable to the JTAG method, which requires the CPU to run at half speed, and has higher implementation costs.

The paper was presented at the IEE/IEEE Workshop on Real-Time Embedded Systems (RTES) in London, December 2001

3.3.2 Paper B

The hybrid method presented in Paper A is described in more detail in this paper and with the results of measuring the direct and indirect costs of a context-switch.

The paper is a MRTC technical report (02/58).

3.3.3 Paper C

An important property of a program is its potential hit-ratio which depends on the different execution paths. These execution paths depend on input-values, the internal state and timing. An execution path with a very high miss-ratio need not be that with the longest execution time. The path with the worst-case cache miss-ratio (WCCMR) will however require more energy to execute than an execution path with a lower miss-ratio. This paper suggests a static method of finding the execution path which yields the WCCMR and it also estimates the miss-ratio level of this path. It can be used for example to calculate power consumption but can also be used as an input for the compiler to optimize the code. The analysis is performed at machine code level so that no manual annotations are necessary and the result is safe since it is over-estimated.

The paper was presented at the Workshop on Embedded System Codesign (ESCODES) in San José, CA, USA, September 2002. The paper was co-written with Jan Gustafsson.

3.3.4 Paper D

A cache memory makes the instruction execution time variable and due to the penalty which a miss in the cache can yield, its performance can be expected to be lower than that of a system without a cache. If such a code is used in a real-time system, the cache can cause a missed deadline and it is therefore common to disable the cache in hard safety-critical real-time systems. This paper proposes a method of determining the cache miss-ratio threshold level at which a cache memory increase the execution time.

Experimental results on a CPX2000 system showed that the threshold level was so high that it was impossible to reach. One reason for this result is that the use of burst-mode to replace missing cache-lines, has reduced the miss-penalty.

The paper is submitted to Design, Automation and Test in Europe conference (DATE 03), München, Germany

3.4 Other publications

- Filip Sebek. “The state of the art in Cache Memories and Real-Time Systems”. *MRTC Technical report 01/37*, Mälardalen Real-Time Research Centre, Västerås, Sweden, September 2001

Chapter 4

Conclusions

4.1 Summary

A summary of the essential conclusions from this thesis:

- Synthetic workbenches have been successfully used to force a CPX2000 computer system into a worst-case execution time scenario during pre-emption. The software needed to perform the measurement can be used to swap out parts of the pre-empted task code in the cache and by this eliminate intrusion and probe effects.
  - The indirect cache cost of a pre-emption is very high. It has been measured to be more than three times the execution cost of the context-switch functions themselves.
- The CPX2000 computer system without caching will never cause a missed deadline if the instruction caching is enabled. The is because the cache lines are filled in burst mode, which reduces miss-penalty.
- In the same way the worst-case execution time path must be determined to avoid missed deadlines, execution paths with the worst-case cache miss-ratio must be identified to quantify the maximum energy usage. This has been achieved with a fully automated statical method.

4.2 Future work

Most of the questions in section 2.2 have been answered satisfactorily, but there are still aspects and features which deserve further study. The results and conclusions have also raised further questions so suggestions to future work have its place;

- The measurement methodology and the generation of synthetic workbenches could be developed for data caches also. Data caches are however more complex to handle than instructions since they also write back data to the primary memory.
- The calculation of worst-case cache miss-ratio can also be extended to handle temporal locality to determine the boundaries of the cache miss-ratio tighter to the actual.

Paper A

Filip Sebek

Measuring Cache-Related Pre-emption Delay on a Multiprocessor Real-Time System

Presented at the IEE/IEEE Workshop on Real-Time Embedded Systems (RTES)

London, December 2001

Chapter 5

Measuring cache-related pre-emption delay on a multiprocessor real-time system

Abstract

Cache memories in real-time systems can increase performance, but at the cost of unpredictable behavior which gives loose bounds on the worst case execution time analysis. Task pre-emption causes a swap of cache contents with an initial performance dip that is considered as a delay. This delay is necessary in execution time analysis and must be added to each task-switch to determine if the task sets are schedulable.

Cache performance and costs have traditionally been estimated through trace-driven simulations, but since representative traces and a true simulation models are hard to accomplish, a “physical” measurement of the system might be the only way to determine its status.

This paper suggests two methods to measure the cache-related pre-emption delay on a Power PC750 multiprocessor system by using the processors’ built-in performance monitor. One method is completely hardware-based and the other has a minimal software support. Both methods pass information to an

external monitor system that stores data with timestamps in a database for further analysis.

5.1 Introduction and motivation

Cache memories are common in computer today's systems to bridge the response time from primary memory which boosts up performance. Since the small cache is too small to hold all data and instructions, blocks are swapped in and out depending on what section of code in the program is handled at any moment. The swapping results to in a variable memory access time, which makes execution time analysis in real-time systems very tricky to analyze.

There are two categories of cache block swap-outs [AHH89];

- Intrinsic (inter-task) behavior depends on the internal design and execution path of the task. Two functions or data areas in the task may compete for the same cache space with cache misses and a resulting performance loss. Increasing the cache memory size or associativity can reduce these effects.
- Extrinsic (intra-task) behavior depends on the environment and the others tasks' inter-task behavior. At context-switch the cache contents will be more or less displaced by the new running task. This performance loss is also called *cache-related pre-emption delay* (CRPD). To eliminate the CRPD and extrinsic cache effects, Kirk suggests partitioning the cache into segments and assign task their own segment of the cache [Kir89]. This can also be accomplished in software by locating code and data so they won't map and compete for the same areas in the cache [Wol93].

Both these problems must be solved or correctly analyzed to be able to give an accurate and tight *Worst Case Execution Time* (WCET). The WCET is the base for all scheduling of a task-set — without it one cannot determine if the task-set is schedulable or not in a real-time system.

Basumallick and Nilsen identify the CRPD in the Real-Time environment in [BN94] with the formula $C' = C + 2\delta + \gamma$, where C' is the new WCET, C stands for the unmodified WCET, δ is the execution time for the operating system to make a context-switch (two are needed for a pre-emption) and γ symbolize the maximum cache-related cost by a pre-emption.

Even though recent research and many different analysis methods have been able to bound WCET tighter with many different solutions, they are still not applicable to industrial systems due to their limitations [TD00, LMW99, HAM$^+$99, TFW00]. A safe approach would assume that the complete cache must be reloaded on a context-switch, but since this shouldn't be possible in, for instance, a system with small but many tasks, the schedule would lead to an underutilization of CPU resources. Today's industrial developers are in a great need of real values to implement new software based products on high-performance processors.

To the best of our knowledge the only work that has been presented to measure the CRPD is Mogul and Borg's trace driven simulation of a UNIX-system [MB91]. Mogul and Borg measured the delay (γ) to $200 - 400\mu s$ of a task, however the traces were not taken from a real-time system and all the time-slices were of equal size. The cache memories of today are larger and more complex than those which were used in these simulations.

Performance estimation on cache memories has traditionally been made with trace-driven simulation. The simulations are mainly of single programs or tasks and with absence of operating system, pipelining, complex cache structures, prefetching features etc. This paper presents methods for measuring the CRPD on a real running multi-processor system.

5.2 The multi-processor system

SARA — Scaleable Architecture for Real-Time Applications — is a research project with a Motorola Compact PCI backplane bus with Power PC750-processor boards [LKF99]. See figure 5.1.

A special *master card* is equipped with a Real Time Unit (RTU)[FSLA95] that controls the execution of the tasks on all processor cards. The RTU is a high performance and performance predictable hardware implementation of an operating system that handles scheduling and other real-time operating system services. No other software is needed. The other processor cards are used as *slaves* to increase application performance. All communication between tasks (inter and intra-processor) is performed through a *virtual bus* which simplifies application development[NL00]. A special device called *Multipurpose Application Monitor (MAMon)* [ES01] is connected to the RTU.

Today, MAMon and the RTU co-exist in the same FPGA, and besides increased performance it is a very practical and cost-effective way to eliminate problems with PCB-layout and other hardware manufacturing issues.

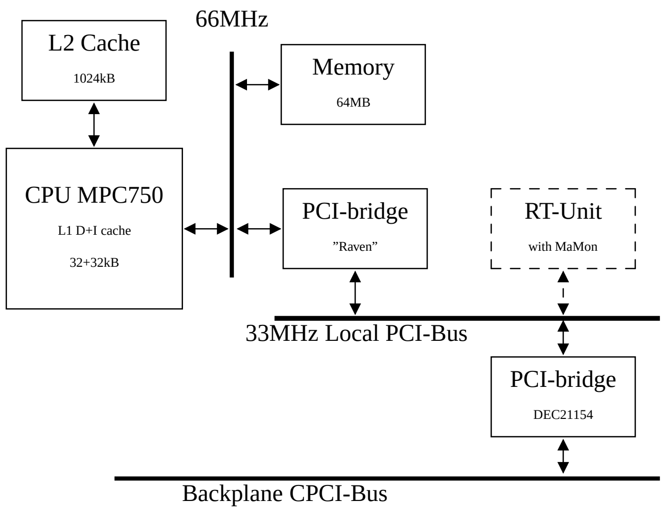

Figure 5.1: CPU-card. The RT-Unit is only on master cards.

5.3 The measurement

The Motorola PowerPC 750-processor (MPC750) is equipped with a performance monitor [Mot97, Mot99] with four dedicated registers which count predefined events such as cache misses, miss predicted branches, number of fetched instructions, and other occurrences. The monitor function is meant to be used to tune performance on software applications and to help system developers to debug their systems.

It is not possible to measure the CRPD time directly since the MPC750 performance monitor is not that advanced. The performance monitor at the processor is set to count cache misses and instruction fetches, which is the information needed to calculate instruction miss ratio. To calculate the data miss ratio, the data misses, and the number of loads and stores must be counted. By continuously measuring the miss ratio, the CRPD can be calculated and presented.

One problem is to distinguish the extrinsic misses from the intrinsic, which is impossible to do exactly, with the suggested measuring model since the performance monitor is unable to categorize the types of the misses. Our approach is to determine the *average miss-ratio-level* of the task and subtract it from the miss-ratio after the context-switch. Figure 5.2 illustrates a scenario with a context-switch from a task with 22 percent average miss-ratio to another task with 12 percent.

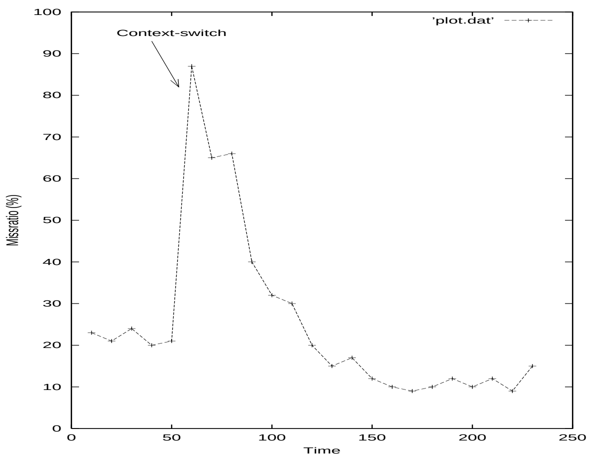

Figure 5.2: Miss-ratio during a context-switch.

5.3.1 With software support

A small, simple, cyclic task — ”MonPoll” — polls the MPC750’s performance monitor registers and passes them to MAMon where they get time-stamped. An alternative solution is to store the data in the task’s local memory, but the writing to memory would compete with the application’s need for cache memory with a serious swap of cache data as a result. A third possibility is to stream the data through an Ethernet card as UDP-messages to a remote host on the network, but these actions lead to substantially longer execution time and a swap of contents in the instruction cache.

MAMon is non-intrusive and provides the needed service with a minimum of software code. Only one simple C-code assignment is all that is needed to store a value in a MAMon register. MAMon sends (through a parallel port) timestamps, register values and other predefined task information to a database

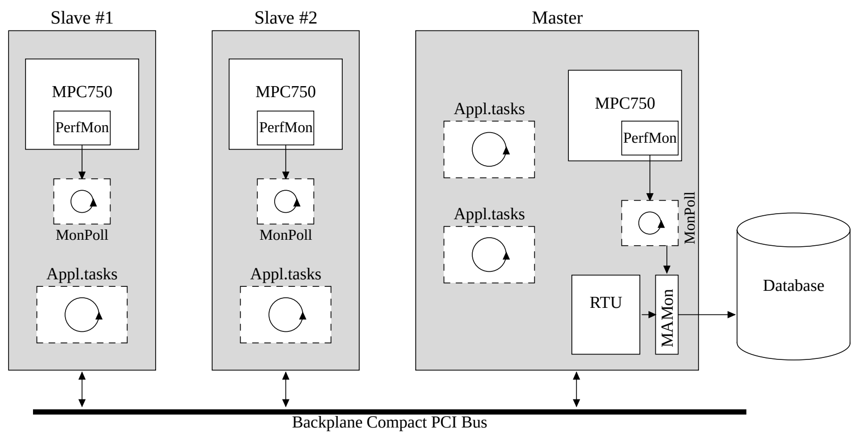

Figure 5.3: SARA system. Dashed boxes are software implementations.

on an external host where it is stored for further analysis and graphical presentation. See figure 5.3.

Even the smallest possible MonPoll-task will, however, interfere with and pollute the cache result by its own presence in the executing environment. It will also use system resources and decrease performance by increasing the execution load. The cache content could remain unaffected by locking the cache memories during the polling, but that would on the other hand have an influence on the execution time.

Leaving the polling task “as is” in the running system can ”solve” the problem. To make a system safe and act as it did during the test phase of its development, probes must never be removed [Tha00]. The cost of having an observable system is to dedicate some percentage of the capacity to software probes. The less frequently that probing is performed, the less interference there is, but with a loss of valuable data. If the performance loss is unacceptable in a running system, the ”MonPoll”-task must be removed. Furthermore, to get a more correct performance value, measurements at different sample frequencies must be performed. By extrapolating these values, more accurate performance estimations will be at hand. This method is applicable on soft real-time systems due to its more or less polluting nature.

5.3.2 Completely in hardware

The register values from the performance monitor can be read through special pins on the processor called JTAG-pins (defined by IEEE 1149). To interpret those signals on an oscilloscope by "hand" is a not a trivial task due to the large amount of information presented in a simple wave-diagram. To cope with this translation and interpretation problem, a special hardware device communicates with the performance monitor through JTAG and "downloads" the requested data. This device could then pass the information further on to MAMon through either dedicated peer-to-peer links or a special bus with MAMon and those devices attached to it. The device acts exactly as the "MonPoll" task described in the previous section but in this case it is implemented in hardware. This method might be suitable for hard real-time applications since it doesn't interfere with the caches or the execution time.

Many hardware constructions cannot run at full speed when JTAG is used, which is true for some CPUs in the Motorola PPC-family. The advantage of this approach is its non-polluting and non-interfering measuring, but the drawbacks are, as mentioned, higher hardware costs and an underutilization of CPU performance.

5.4 A synthetic workbench

No good standard benchmark are available today to measure cache memory effects in real-time systems. Non-real-time benchmarks such as SPEC or Dhrystone are just single programs without (interfering) tasks. Rhealstone[Kar90], on the other hand, simply tests real-time operating system issues such as task-switches, deadlock handling and task communication. The test applications are too small to test cache memory issues and were not meant to do so either.

The test will therefore be performed on synthetically generated task-sets where the amount of tasks and the data and instruction size of all the tasks will be generated by a special program. Task interaction, priorities, and cycle time can also be set. By altering the mentioned parameters and measuring the hit ratio over time, a pretty good view of the CRPD will be available. This method has been used successfully in, for example, Busquets-Mataix *et al*'s work[BMWS$^+$96].

5.5 Conclusions and Future work

Performance estimation on cache memories has traditionally been made with trace-driven simulation with only pieces of a complete trace and simplified simulators. To be able to make a correct and tight worst-case execution time analysis all types of delays and interference in the execution must be identified. One such property is the Cache-Related Pre-emption Delay (CRPD), which is a product of extrinsic cache behavior.

This paper suggests two methods to measure hit-ratio by using the processors built-in performance monitor. Either a software task or a hardware device can poll the data from the performance monitor. The software solution is performed with a minimum of code to reduce a probe effect. The polled information is then passed to a system monitor that collects and stores context-switch information and other predefined events in a database. Since all events are time-stamped, the miss-ratio can be continuously monitored and the CRPD can be calculated. The methods will give applicable values to execution time analysis since they measure real time on a real running system. A complete hardware solution might be better since it could be intrusive free, but the data can only be available when the processor is running at half speed if JTAG is used.

Standard benchmarks don't cause pre-emption and that is why CRPD cannot be measured at those. Generated task-sets with synthetic tasks of different numbers, sizes, relations, cycle time etc. will generate pre-emptions to be measured and will also give a figure which shows how exact the suggested measurement method is.

Due to heavy performance loss and costly redesign of PCB-layout, the hardware device solution is excluded and only the software solution is motivated to implement in the future work.

Paper B

Filip Sebek

The real cost of task pre-emptions
— measuring real-time related cache performance with a hw/sw hybrid technique.

Technical report 02/57, Mälardalen Real-Time Research Centre, August 2002.

Chapter 6

The real cost of task pre-emptions — measuring real-time-related cache performance with a HW/SW hybrid technique

Abstract

Cache structures and other modern hardware is so complex today, that simulation on the instruction level is very complicated and time-consuming. Real measurement is much faster, but with the disadvantage of being less observable and, in most cases, impossible to make non-intrusive.

To predict shedulability for a system that incorporates an unpredictable device, such as the cache, requires knowning data such as the task execution time, task-switch time, and pre-emption delay to statically predict their schedulability. This paper proposes a hybrid HW/SW method to measure cache performance with minimum intrusion. It also presents some experiences with a real system and the experimental results of setting up scenarios to measure cache performance on a high performance microprocessor system based on MPC750 CPUs.

The experimental results show that the cache-related pre-emption delay for instructions can be up to 69% of the pre-emption cost for a MPC750 system.

6.1 Introduction

6.1.1 Real-time and cache memories

Cache memories are used in today's computer systems to boost up performance by bridging the gap between response time of primary memory and CPU. Since the small cache is too small to hold all data and instructions, blocks are swapped in and out depending on what section of code in the program is handled at the moment. The swapping results in a variable memory access time, which makes the computation of the worst-case execution time (WCET) very tricky. The execution time depends of the cache contents and the cache contents depends on the execution path that depends on, among other properties, the execution time.

System developers are today in a great need of real values to implement new software based products on high-performance processors. These values must be safe but also simple and fast to get. A correctly performed real measurement on a real system might give them what they need — especially when the offered method is easy to understand.

6.1.2 Monitor and measurement methods

Several methods are feasible to measure or monitor cache performance in computer systems. The major issue is to make the measurement non-intrusive so the measured environment is unaffected. A second issue is to set up correct and representative scenarios to be measured. If for instance the worst-case execution time (WCET) is to be measured, one must set up an execution path that leads to the WCET.

Execution time and other performance issues can either be statically analyzed [HAM$^+$99, TD00, FMWA99, LHM$^+$97] or simulated[MB91, SL88], or measured directly on the target system[LHM$^+$97, PF99].

The advantage of static methods is that they are safe if the system model and analysis method are correct and compatible with each other. The hard part is to add complex structures into the model like pipelining, cache memories, DMA and other hardware that affects the execution time. To model a real processor is very difficult to accomplish[AKP01, Eng01] and to simulate execution on a modeled complex system may take over 1000 times longer than the actual execution.

Real measurement is on the other hand much faster to perform, but requires special hardware to tap information from the system. Changing hardware parameters like cache size or bus bandwidth is often practically not possible. The major advantage is that a complex hardware is correctly "modeled" as is.

Monitoring methods can be categorized as

- **Trace driven simulation.** By feeding extracted execution traces to a simulator it is possible to monitor internal states and measure execution time. It is also common that architecture parameters (cache size, band width etc) can be altered to give a better flexibility in the analysis. One problem arises if the execution path (i.e. the trace) will change due to timing of input data. Since the execution time depends on the execution path and cache contents, the conclusion is that the simulation must model the system from where the traces were derived.
- **Hardware monitoring.** By attaching for instance a logic analyzer on the bus or taping information from the processor's JTAG pins one could trace where the execution occurs. One problem is that JTAG has a limitation in bandwidth so for instance a processor must run at half speed. Another problem is that only addresses on the address bus doesn't say much about what really is going on in the processor since the information is at a very low level and internal signals to registers and caches are hidden.
- **Software monitoring.** A program or pieces of code writes internal states or time values to memory, disc or screen. The information can be at a very high abstraction level, but the cost is a high utilization of CPU, memory, cache alteration, bus bandwidth etc. All these resources must be allocated and remain in the program after the measurement to eliminate probe effects.
- **Hybrid HW/SW monitoring**. With minimal code and hardware monitoring it is possible to get information at a high level with small resource utilization.

To measure with the SW or SW/HW monitoring methods, *probes* are inserted in the code. The methods proposed in this paper use three kinds of probes: *Kernel probes* that are incorporated in the OS-code of task-switches, system functions, interrupt routines etc. *Inline probes* are placed directly in the application code and *probe tasks* that log the system state.

6.1.3 Cache properties to measure

In a real-time perspective some properties are more interesting and useable to measure than others.

- **Task-switch time**. It is very common in schedulability analysis to just take the WCET for the tasks as parameters and assume that the context-switch time is zero. In the real world a time will elapse during the context switch since the OS has to execute some code to make the context-switch possible. The knowledge of this time is important in systems with a high frequency of context-switches.

- **Pre-emption delay including cache-related effects.** The cache will be filled with instructions and data that belong to the executing task. Misses that occur at this point are called *intrinsic*. When a new task starts to execute, the cache will swap out the previous task with the current locality — these kinds of misses are called *extrinsic*[AHH89]. One special kind of extrinsic miss occurs during pre-emption when a high prioritized task can interrupt low prioritized to gain faster response time. The pre-emption will swap out the cache contents of the executing task and cause a cache refill-penalty for the low prioritized task when it resumes its execution. The cost to pre-empt another task is $C' = C + 2\delta + \gamma$, where C' is the new WCET, C stands for the unmodified WCET, δ is the execution time for the operating system to make a context-switch (two are needed for a pre-emption) and γ symbolizes the maximum cache-related cost by a pre-emption[BN94]. This *cache-related pre-emption delay* (CRPD) or *cache refill penalty* that can be considered as an *indirect cost* is illustrated in figure 6.1 and 6.8.

  The CRPD can be eliminated or reduced by partitioning the cache so each task has a private part of it, but to the price of decreased over-all performance[Kir89, Wol93, Mue95].

- **Continuously measuring cache performance**. By continously monitoring the cache-miss ratio, maximum and average miss-ratio during a time-slice can be determined. If the time-slice is smaller than the operating system's time granularity, the average miss-ratio can be used to calculate WCET and if used in soft real-time systems the maximum miss-ratio can be used. The method is however only applicable for hard real-time systems when the worst-case execution time scenario is executed.

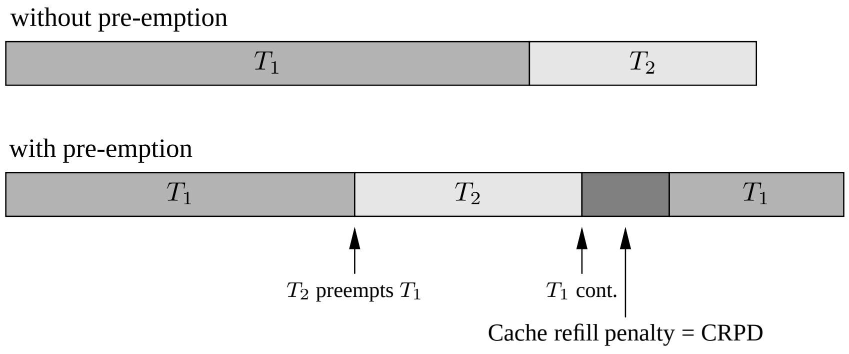

Figure 6.1: Cache-related pre-emption delay. A pre-empted task can be drained by cached data and suffer a refill penalty. Note that the penalty does not come right after resuming the pre-emption — it may come later or in pieces depending on the program's design.

This paper is organized as follows: The next section describes the target system where all measurements were performed. Section 3 presents the workbenches in detail and experimental results, and the paper ends in Section 4 with conclusions.

6.2 The system

The complex target system "SARA"[LKF99] has been used for experimental results and testing the measuring method. This section will describe features and special hardware components in the system. Please observe that all the features in the hardware is not necessary for the generalized measuring method — it just makes the method easier to perform.

SARA — Scaleable Architecture for Real-Time Applications — is a research project with a Motorola CPX2000 Compact PCI backplane bus with at most eight Motorola Power PC750-processor (MPC750) boards [LKF99]. See figure 6.2 but also figure 6.4 for a complete system overview. The processor boards are equipped with a MPC750 running at 367MHz and is connected to a 66MHz bus as well as the the 64MB DRAM main memory.

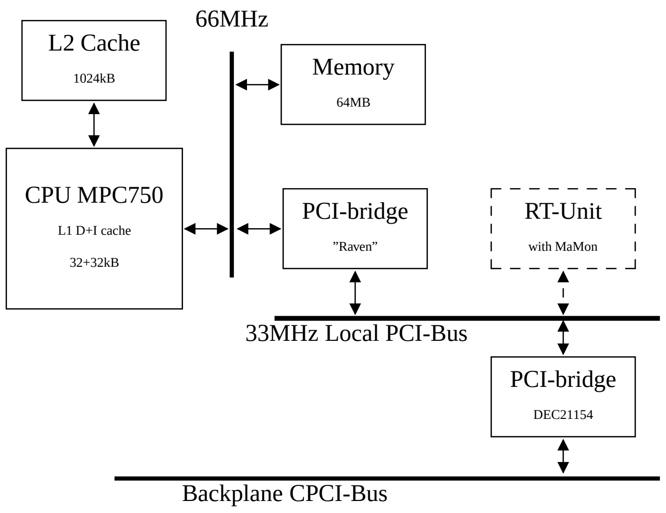

Figure 6.2: CPU-card. The RT-Unit is only on master cards.

6.2.1 MPC750

The microprocessor MPC750 is equipped with a splitted instruction and data cache at the first level, and a unified cache memory at the second level. The first level caches are 32kB each and organized in 8-way set-association with a pseudo LRU replacement policy. Each cache block can hold eight 32-bit words. The second level cache is in the target system 1024kB, 2-way set-associative and the block size is in this case 128 byte that is divided into sub-blocks. The caches are non-blocking and can be locked by users during execution. Four areas in the memory that is to be cached are defined in Block Address Translation (BAT) registers. All four areas can set its own WIMG-properties, that is **W**rite-through/Write-back policy, caching **I**nhibited, enforced **M**emory coherency and **G**uarded bits.

The MPC750 is equipped with an on-chip performance monitor that can monitor 48 different kind of events, but there are only 4 performance monitor counters (PMCs) available. Only 5 of the 48 events can be associated with any PMC and the rest are associated to a dedicated PMC. [Mot97, Mot99]

6.2.2 Real-Time Unit

A special *master card* is equipped with a Real Time Unit (RTU)[FSLA95] that controls the execution of the tasks on all processor cards. The RTU is a high performance and performance predictable hardware implementation of an operating system kernel that handles scheduling and other real-time operating system services. The other processor cards are used as *slaves* to increase application performance. All communication between tasks (inter and intra-processor) is performed through a *virtual bus* which simplifies application development[NL00].

6.2.3 MAMon — an application monitor

A special device called *Multipurpose Application Monitor (MAMon)* [ES01, ES02], can tap for instance the RTU non-intrusively on information regarding context-switching, inter process communication, task synchronization etc. There is also a possibility to write to special registers called *software probes* directly from the application. Writing to a software probe is much faster than reading a register since the processor just writes to the PCI-bridge through the 66MHz 60x-bus. The bridge will then eventually write to the RTU when the 33MHz local bus is clear.

All the collected data is sent through a parallell port to an external host for post analysis.

Today, MAMon and the RTU co-exist in the same FPGA, and besides increased performance this is a very practical and cost effective way to eliminate problems with PCB-layout and other hardware manufacturing issues.

To integrate MAMon into another system than SARA that doesn't use a hardware implemented OS can easily be performed by just adding a card with MAMon hardware, which is accessed by memory mapped addressing. In this case only software probes can be used.

6.3 Experimental setup and results

All experiments were performed at instructions only with synthetic workloads on the SARA system. Each of the three measurement cases are presented in their subsection. The first case describes how to measure cache miss-ratio continuously with minimal intrusion. The second case describes how to measure context-switch/pre-emption time and the third case sets up a scenario to maximize cache effects so the cache-related pre-emption delay (CRPD) can be measured and included into the pre-emption time. The subsections also presents the measured results and concludes with a discussion.

6.3.1 Synthetic workloads

No good standard benchmark suits are available today to measure cache memory effects in real-time systems. Non-real-time benchmarks such as SPEC or Dhrystone are just single programs without (interfering) tasks. Rhealstone[Kar90] on the other hand just tests real-time operating system issues such as task-switches, deadlock handling and task communication. The test applications are too small to test cache memory issues and were not meant to do so either.

The tests were therefore performed on synthetically generated task-sets where the amount of tasks and the data and instruction size of all the tasks were generated. Priorities, miss-ratio, cache locking and cycle time can also be set. The generated code is very simple since it's only purpose is to swap out cache contents. The basic structure of a task is one simple big loop that contains a large sequence of "r1=r1+0" without jumps and ends with a delay to let lower prioritized tasks to run.

Using synthetic task-sets has successfully been used in for instance Busquets-Mataix *et al*'s work[BMWS^{+}96]. More about the synthetic workload and its different properties will be presented in the three cases.

6.3.2 Continuous measurement

6.3.2.1 Implement the probe as a task

Implementation A small, simple, cyclic probe task — "MonPoll" — polls the MPC750's performance monitor registers and passes the values to MAMon where they get time stamped. Figure 6.4 illustrates the complete system with hardware and software. This small task (written in C) is as simple as in Figure 6.3.

The task should have the highest priority to be able to measure during all tasks' execution.

Performance requirement The OS[1] granularity to start tasks is 2 milliseconds, which means that the highest sample rate is 500Hz. The task requires

[1]"OS" and "RTU" will in this case referred as the same thing.

```
void monitor_poller( void ){

  RTU_IO rtu_io;

  while(1) {
    asm(" mfspr 0, 938 ": "=r" (MAMON_SWPROBE_2));
    asm(" mfspr 0, 941 ": "=r" (MAMON_SWPROBE_3));
    rtu_io.delay(2);
  }
}
```

Figure 6.3: A small task that polls performance monitor counters and writes the results to MAMon

671 assembly instructions to execute which includes two context-switches in the operating system (major part) and the small code itself.

Best-case is when no cache misses are present. The execution time for the MonPoll task is then 43.1 μs , which means that it consumes about 2% of the execution time. It also means that if a task is interrupted by the MonPoll task and no cache misses will occur, the execution time of the running task will be extended with 43.1 μs .

Worst-case will occur if the MonPoll task is not in the cache and will pre-empt a loop for which content maps the same cache lines as the MonPoll task. This will generate a small burst of initial misses for the MonPoll task plus some new misses when the pre-empted task resumes its execution to replace the MonPoll task in the cache (=CRPD).

Specifically this means an extension of the execution time by 21.9 μs to 65 μs . 99 cache lines have been swapped out and the refill time for those is also 21.9 μs if all cache blocks were useful in the pre-empted task. The performance cost will therefore be more than doubled from 43.1 to 86.9 μs , or if the sampling is performed at maximum rate $\frac{0.0869}{2} \approx 4\%$ of CPU resources.

Analysis and Discussion A solution to reduce the performance cost of the measurement method is to keep the MonPoll task in the cache, which can be achieved by (software) partitioning. The cost is high because it allocates a

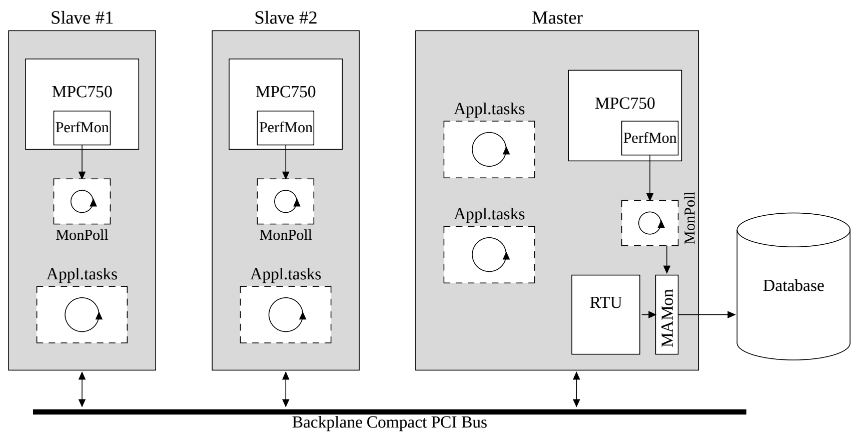

Figure 6.4: SARA system. Dashed boxes are software implementations.

task with all the inherited costs of context switches etc. Only a few registers will be used and therefore it is very expensive to store all those only to make a simple poll. The granularity of the observation by only being able to poll each second millisecond might be very poor in many situations. The MPC750 will execute almost one million instructions during this period.

6.3.2.2 Implement the probe as an interrupt routine

Instead of putting the small piece of code into a task, the same code can be called as an exception routine. There are several pros and cons to do so:

- Saving registers before running own code must be performed by "user" instead of the operating system
- It is performed by the processor itself, no operating system support is needed which also means faster execution and less intrusion
- Finer granularity is possible since the interrupts are independent of the operating system's time base

Exception on timer value The MPC-family is equipped with a 32-bit decrement-register (DR) that is decreased by a step each fourth external bus-cycle which in this specific case is 66,7/4 MHz = 16.7 MHz or T=60ns. When DR is equal to zero an exception occurs and the program counter is set to 0x0900. If this feature is unused this address must contain `rfi` (return from interrupt) to proceed with the execution. DR can be set to any arbitrary number by user code and by this be used as an external clock. The CPU runs at 233MHz which means that a resolution of 14 clock cycles or at maximum 28 instructions is possible to achieve.

MAMon is able to handle about 3500 cache events per second. Sending performance monitor data at this pace is a severe limitation; during this period 150 000 instructions may have been executed. A workaround is to write to MAMon only when an amount of changes (for instance cache misses) has reached a limit. Since MAMon can store short bursts of data in a FIFO queue, a practical sample rate of up to 1MHz is possible but then the load at the system will be 12%[2]

Exception on PMC threshold value The MPC750 has a special exception routine for the performance monitor. When a PMC reaches the threshold value an exception call to 0x0f00 is performed. In this case the workaround in the previous timer value solution can be avoided with a performance increase and less intrusion as a result.

6.3.3 Workbench

The generation of code with a fixed miss-ratio has been accomplished in two ways. Either spatial locality is exploited by only executing a fix fraction of the instructions in a cache line or a fraction of reuseable code is used to exploit temporal locality. A third possibility is to combine both these methods.

- If the first instruction in all cache lines is an unconditional jump to the start of the next cache line only the first word in each block will be accessed and by this never exploit spatial locality. If the code mass is much larger than the cache memory the reuse of code will not take place and decrease miss-ratio. By this we have generated a code with 100% cache misses. To generate code with 50% misses the jump is moved from the first to the second word in the cache block, and to generate code with 33% the jump is moved to the third word and so on. 66% misses can

[2]The 12% load is best case when no cache misses occur. This value shouldn't be compared with the MonPoll 4% utlization since that only runs at 500 Hz.

$$miss\ ratio = \begin{cases} 0\%, & ts \leq cs \ \wedge \ time \rightarrow \infty; \\ \frac{ts-cs}{cs} \cdot set\ associativity, & cs < ts < cs + \frac{cs}{set\ associativity}; \\ 100\%, & ts \geq cs + \frac{cs}{set\ associativity}; \end{cases}$$

Figure 6.5: The cache miss-ratio depends in the generated code on task size, cache size and set-associativity. Each cache block begins with an unconditional jump to the next cache block to prevent utilization of spatial locality. (*ts* = task size, *cs* = cache size)

be generated by altering the jump from the first and second word in the cache block in the complete program.

- Code within a loop that fits into the cache will gain a lower miss-ratio each iteration since it reuses the cache lines (temporal locaility). If the code is a bit larger than the cache size some code will be swapped out and the amount of *useable cache lines* in the cache will decrease. With code that has 100% spatial misses the cache miss-ratio is formulated in Figure 6.5.

Example: A 38kB task with 100% spatial misses will in a 32kB 2-way set-associative cache memory obtain an average miss-ratio of $\frac{38-32}{32} \cdot 2 = 37.5\%$

A more correct description of “task size” is “the task’s cache-non-interfering *active* or *useful* cache lines” but in this synthetic generated workload it is the same.

6.3.3.1 Results

To prove that the measurement methods work (both probe task and interrupt driven kernel probes) a task-set with tasks with different average cache miss-ratios was set up, executed and measured. The theoretical values were compared with the measured and the difference was less than 1% with the interrupt-driven measurement method. The fluctuation depends for instance on DRAM refreshment and odd bus-cycle access.

Figure 6.6 illustrates four tasks with different miss-ratios executing. The tasks’ miss-ratio was controlled by the individual task size.

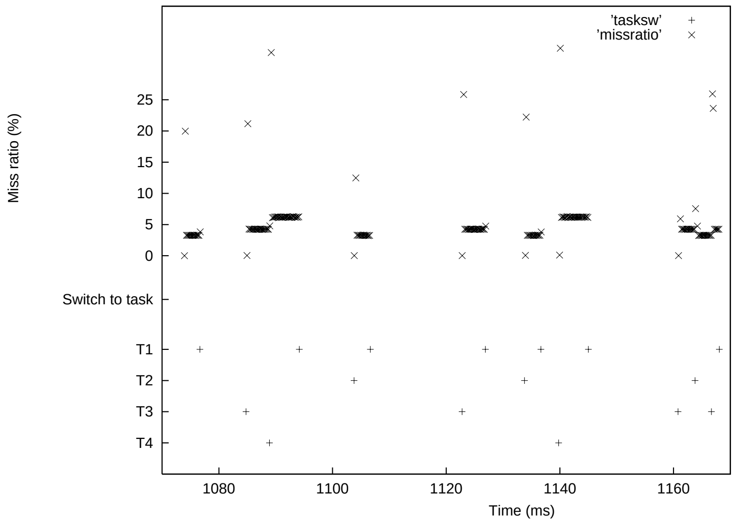

Figure 6.6: Continous miss-ratio measurement with an interrupt driven kernel probe. T_2 starts to execute (miss ratio: 3.27%) and yields to the idle task T_1 (0%). Then T_3 (4.25%) executes and yields for T_4 (6.20%). At the end of this sample T_2 pre-empts T_3. Observe the inital miss-ratio peaks at the context-switches.

6.3.3.2 Discussion and conclusion

This kind of monitoring is maybe more of performance than of hard real-time interest. The monitoring has too low resolution to give any information about for instance CRPD. It can however be used for *soft* real-time systems to get a view of the *average* miss ratio of the tasks to be used for static WCET-calculation.

A periodic interrupt routine with inline probes has shown much better performance than a probe task. The granularity can increase from 500 Hz to 1 MHz in short[3] bursts. CPU utilization with a probe task running at 500Hz drops from 4% to an immeasurable level if the same measurement is performed with the interrupt routine implementation running at the same frequency. When

[3]"Short" in this case are 256 events since that is the size of the internal FIFO-queue in MAMon. Availbale hardware sets the limit of the queue size.

measured at 1 MHz with the interrupt driven method the CPU used 12% of its computation resources to the measuring activity.

6.3.4 Measuring context-switch time

A pre-emption consists of two context-switches:

1. Interrupting the executing task.
    (a) OS decides to make a context switch
    (b) registers of the pre-empted task are saved
    (c) registers of the pre-empting task are loaded
2. Resuming to the interrupted task.
    (a) the high prioritized task yields to low prioritized tasks
    (b) OS decides what lower task should run
    (c) registers of the high prioritized task are saved
    (d) registers of the lower prioritized task are loaded

Since the RTU on the target system does very much of the OS work during the interrupt and more software code is to be executed during the resuming, it is expected that the resuming in this case should take longer time.

6.3.4.1 Workbench and results

This type of measuring is quite straightforward since four kernel probes into the OS can perform it. The probes are placed at the first and last lines of the two context-switch routines. The best case scenario is when the routines are in the cache and the worst-case is when all the code has to be loaded from main memory and swaps out some cache lines that would have been useful for the task (=CRPD).

Best case scenario is created with an empty task “pre-empting” the idle task. The worst-case scenario is measured on a task-set with two tasks that both are larger than the cache memory so the OS-code will be swapped out for sure. The results are presented in Table 6.1

Sitaution	Number of cache misses	Interrupt (μs)	Resume (μs)	Pre-emption cost (incl. CRPD) (μs)
Best-case	0	18.5	24.6	43.1+0.0=43.1
Worst-case	44+55	28.4	36.6	65.0+21.9=86.9
No cache	-	52.5	66.1	118.6

Table 6.1: The table shows the time it takes to perform a context-switch. The cache-related costs are inherited only from the context-switch code.

6.3.4.2 Discussion and conclusion

Even if the context-switch time relies on many parameters such as CPU, system platform, operating system, and in this case it is more true than for others since parts of the OS kernel is implemented in hardware, the proposed measurement method is still useable for almost any computer system. The measured time can directly be used in scheduling algorithms and analysis.

6.3.5 Measuring CRPD

6.3.5.1 Workbench

The CRPD grows linearly with the pre-empted task's size[4] and reaches its maximum value when the task size is equal to the cache size assuming that the pre-empting task has replaced the suspended task completely.

One should notify that the CRPD will decrease after its maximum point since the number of useful blocks will decrease with the task's size. If the active context of the program is twice as large as the cache size it will never reuse any cache lines. In this case there will be no CRPD what so ever.

On set-associative caches with LRU or FIFO replacement algorithm the CRPD also will depend on the set-associativity; a fully associative cache memory will with a task that is one word larger than the cache always replace the cache line that is about to be accessed and by this never have useful cache-lines in the cache — poor performance but no CRPD. The relationship between CRPD, task size and set-associativity is illustrated in Figure 6.7 and the reasoning is similar to generation of a task with fix miss-ratio (section 6.3.3).

[4]As mentioned before; a more correct description of "task size" in this context is "the task's cache-non-interfering *active* or *useful* cache lines".

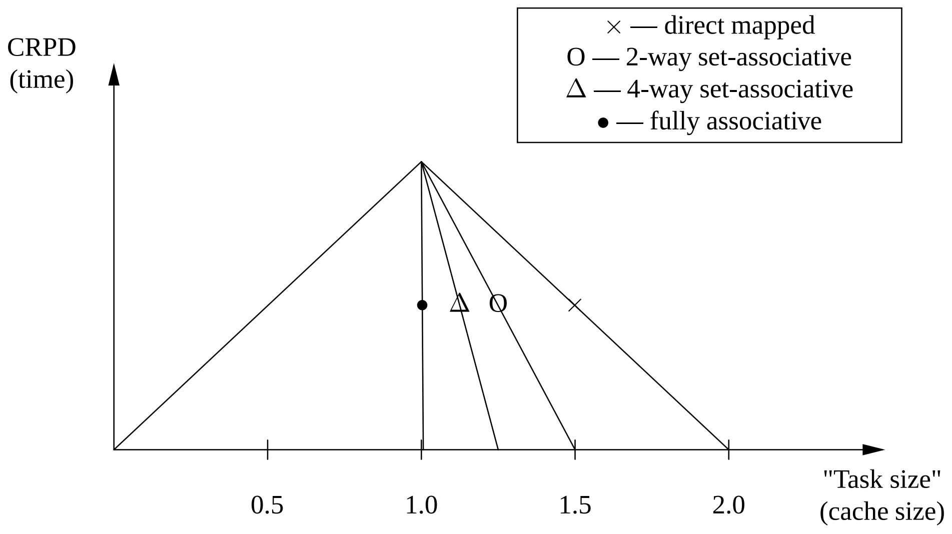

Figure 6.7: The theoretical maximum CRPD of a task is depending on the tasks' size and the cache memory's set-associativity.

Figure 6.8 illustrates the scenario where the high prioritized task T_2 pre-empts T_1 and by this get CRPD=((f-e)+(d-c))-(b-a).

To measure the *maximum* CRPD that is possible to suffer in a system, the previous scenario can however be simplified by pre-empting a task (T_1) by another task (T_2) for which both sizes are exactly equal to the cache memory. (T_1) is an endless loop that writes to a timestamped software probe each iteration to make it possible to calculate the execution time. If there are no other interfering components (other interrupts, instruction pipeline refill etc.) the CRPD is at hand. Interfering components can be detected by measuring the CRPD with different sizes of (T_1) since it in the absence of other pre-emption delay components should grow linearly with a start in the origo, and reach the top at the cache memory size.

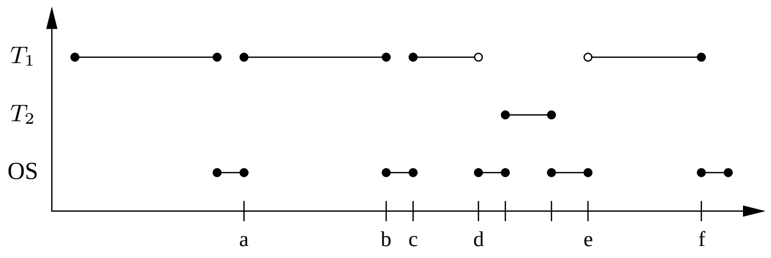

Figure 6.8: During the scenario T_2 pre-empts T_1 and interferes in the cache partly or completely. The CRPD effect will be maximized if all cachelines are useable for T_1 and swapped out by T_2. CRPD = ((f-e) + (d-c)) - (b-a)

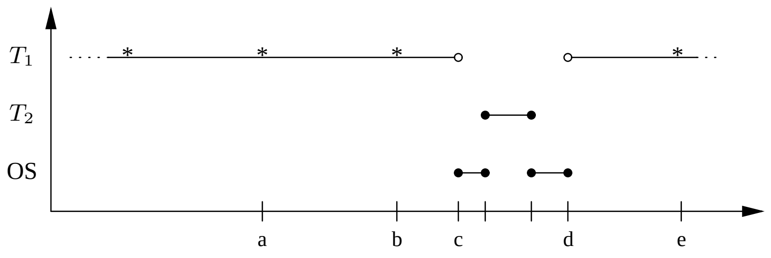

Figure 6.9: A scenario to get the maxmimum CRPD in a system: T_2 pre-empts T_1 and interferes in the cache partly or completely. T_1 is running in an endless loop and timestamps each iteration (marked as ’*’) in the figure.
CRPD = ((e-d) + (c-b)) - (b-a)

6.3.5.2 Results

Several task-sets as in scenario in Figure 6.9 with different sizes of T_1 and the result is shown in Figure 6.10. The maximum CRPD was measured to 195.5μs and the sloping line after 100% task size intersects the x-axis at 113.6% which is less than 1% overestimation from the theoretical 112.5% (see Figure 6.7). The measurement was also performed on a 233MHz MPC750 and since the main memory and busses are the same as on the 367MHz-system the CRPD was the same — a fact that verifies the method's correctness.

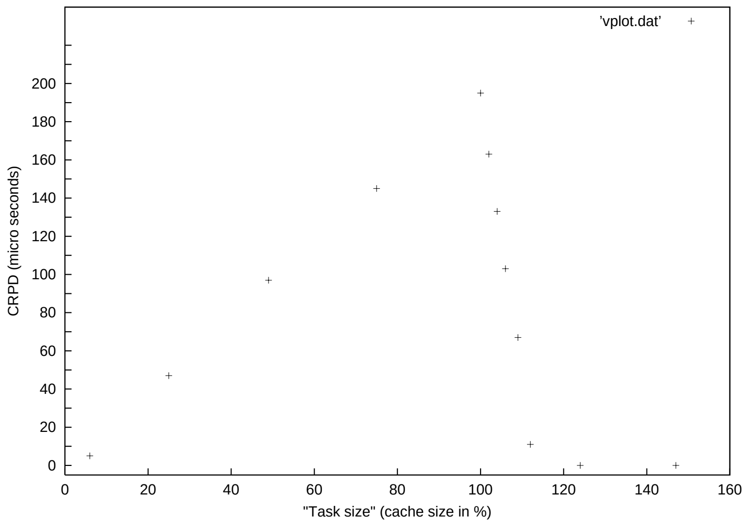

Figure 6.10: Measuring the instruction CRPD on a CPX2000 system with a MPC750 processor that is equipped with a 32kB 8-way set associative L1 instruction cache.

When the 1024kB second level cache was enabled the CRPD was decreased to 31.8 μs . This is however not the *maximum* CRPD of the system since the L2 was only partly used. In this case the L2 could host all tasks and OS-code and no accesses to main memory was necessary.

6.3.5.3 Discussion and conclusion

The CRPD is at most 195.5μs on the considered SARA-system. The executing OS-code to pre-empt a task was measured to 86,9 μs [5], which means that the total pre-emption delay is 282.4μs . In relative terms the major part of the context-switch cost, or $\frac{195.5}{282.4} = 69\%$, is cache-related.

It is quite interesting that the CRPD is almost the same compared to Mogul and Borg's measurements a decade ago[MB91], which were 10-400 μs . Dur-

[5]See worst-case scenario in Section 6.3.2.1

ing this time the processors have become magnitudes times faster and this means that the CRPD has grown in relative terms.

The method to get the CRPD is practicable to get a safe value that is directly useable in a scheduling algorithm. Even if the value is overestimated, the method will never fail or have any limitations that are very common in static methods.

6.4 Conclusions

Real-time systems are often implemented as scheduled tasks with timing constraints that must be satisfied for correct function. Even if very much research has been done in schedulability analysis, it is common to do assumptions to simplify the analysis — for instance that the cost of task pre-emption can be approximated to zero. In reality pre-emption cost, since execution of code takes time, and the indirect refill penalties with cache memories in the system can cost much more. Since the pace of instruction execution accelerates much faster than main memory access time, the penalty has increased and will increase even more in relative terms.

This paper presents a hybrid HW/SW technique that makes it possible to measure and timestamp cache events with minimal intrusion. The method has been implemented on a high-end multiprocessor system with Motorola Power PC750 CPUs controlled by a centralized real-time unit (RTU) with operating system features implemented in hardware. A hardware implemented monitor co-resides with the RTU on the same chip which makes it possible to tap information non-intrusively.

We have proposed how to continuously measure cache miss-ratio with a probe task and interrupt driven kernel probe. The interrupt driven kernel probe outcompetes the task probe in less CPU utilization and finer time granularity.

Methods has in this paper been proposed how to set up workbenches and measure the pre-emption delay including cache-related for instructions. Experimental results showed that a pre-emption could vary from 43 to 282 μs if the cache-related pre-emption delay (CRPD) is counted in. The CRPD itself will in this case stand for 69% of the time of the pre-emption execution and its indirect cause.

Paper C

Filip Sebek and Jan Gustafsson

Determining the Worst-Case Instruction Cache Miss-Ratio

Presented at the IEEE Workshop on Embedded System Codesign (ESCODES)

San José, CA, USA, September 2002.

Chapter 7

Determining the Worst-Case Instruction Cache Miss-Ratio

Abstract

A program with a high cache miss-ratio will lead to longer execution time and a higher power consumption. By knowing the cache miss-ratio, performance can be estimated in advance and can be used as input for compilers and system developers.

This paper presents a method to bound the worst-case instruction cache miss-ratio of a program. The method is static, needs no manual annotations in the code and is safe in the meaning that no under-estimation is possible.

7.1 Motivation

The cache miss-ratio is an important property of a program. The worst-case cache miss-ratio (WCCMR) is useful for instance in the following areas:

- Power-aware systems. The energy consumption grows with higher cache miss-ratio since cache misses leading to more bus and main memory activity [DNVG01].
- Multiprocessor systems on a shared bus. A high cache miss-ratio will congest the bus with a performance loss in the average case.

- Real-time systems. Cache memories are in many cases avoided in real-time systems due to their highly complex behavior. Even if most applications have less than 10% miss-ratio in the instruction cache, there is still a chance that an application can miss more during a certain section of a program. If these bursts of misses occur during a critical phase of the execution, the program might miss its deadline with possibly a malfunction as a result.

One should observe that the execution path with the highest miss-ratio must not be equal to the one with the longest execution path in time or number of instructions executed. A simple example is a polling task that in the average case only makes conditional jumps over a very complicated algorithm that is executed only once a fortnight.

7.2 Related and adjacent work

Much research has been performed to calculate the worst-case execution time (WCET) that is an important property of a task or process in a real-time system where the timing constraints must not be exceeded to have a correct function. Adding caches to a real-time system is a non-trivial task since the execution time will become variable depending if the executing instruction or accessed data is in the cache or not. Some methods have nevertheless successfully been able to make system with caches analyzable [LMW95, AMWH94, LML^{+}94, LS99b, OS97, SA97], but they all aim at WCET-analysis. To the best of our knowledge no one has tried to bound WCCMR, even if many of the WCET-analysis algorithms with some modifications would be able to perform such analysis.

The major difference between the adjacent work and our proposition is the simplicity of our approach.

7.3 The concept and approach

When the CPU fetches a new instruction or data, and it is not in the cache memory, a *miss* occurs and reload from the main memory must be performed. To reduce the penalty, not only the missing instruction is fetched but a complete *line*[1] of instructions is transferred since the main memory can burst consecutive

[1]A cache *line* is sometimes also called *block*, but since the term might be mixed up with for instance *basic block*, this paper will use the term ”line”.

memory locations to the cache. This line might be 2, 4, 8 or 16 words large. The *spatial locality* is exploited which refers to the fact that nearby memory locations are more likely to be accessed than more distant locations.

If, for instance, the line size is 4 words and the program is one sequence of consecutive instructions without branches (”a single basic block”), a miss will occur every 4:th instruction. This situation leads to a miss-ratio of $\frac{1}{4} = 25\%$ and if the line size is 8 words, the miss-ratio will be $\frac{1}{8} = 12.5\%$

If only a part of the instructions in the line will be executed and a jump to another cache line occurs, this line will have a higher miss-ratio. If, for instance, the second instruction in a 8-word-line is a branch, this particular cache line will have a miss-ratio of $\frac{1}{2} = 50\%$. The worst-case scenario is when the first instruction of a cache line is a jump; in this case the miss-ratio is 100%. To suffer from 100% cache misses of a complete program means that it only performs jumps to uncached lines and such program will never be implemented since it cannot do anything useful. The myth of the always cache missing program in a real-time system should hereby be unveiled[Seb02b].

The bottom line of this argumentation is that many jumps and early jumps in a cache line leads to a higher miss-ratio. By analyzing the jumps in a code, the cache miss-ratio can be estimated.

7.3.1 Limitations

The proposed method will not exploit temporal locality and will therefore assume that all new cache line accesses are misses and will only take advantage of the spatial locality. The estimation of the miss-ratio will never be lower than $\frac{1}{linesize}$.

Set-associativity, non-blocking caches, data caches and prefetching are not handled. The analyzed code must always terminate so endless loops are not permitted. A non-terminating process (common in real-time systems) can however be analyzed by removing the “big loop”.

7.4 The algorithm

This section presents the algorithm to calculate the worst-case cache miss ratio in detail.

7.4.1 Overview

1. Construct a Control Flow Graph (CFG) of the program at machine code level.
2. Identify all conditional and unconditional branches and determine their position in the cache line.
3. Identify all jump target addresses and determine their position in the cache line.
4. Calculate all instructions' "local miss-ratio".
5. Determine the maximum and minimum number of iterations in loops (performed at intermediate code level).
6. Construct a binary tree of possible execution paths that might lead to a worst-case cache miss-ratio.
7. Traverse the tree and find the execution path with the highest cache miss-ratio.

7.4.2 A Control Flow Graph

A control flow graph (CFG) [ASU86] describes the possible execution paths through a program. The nodes in a CFG represent *basic blocks*[2], and the edges represent the flow of control.

7.4.3 The "local miss-ratio"

Each instruction is associated to a miss-ratio that is equal to the inversion of the distance between the incoming and the outgoing arrow relatively to the instruction. We will in this paper call this miss-ratio of each instruction for the *"local miss-ratio"*. See the examples in Figure 7.1. The (a) figure illustrates a cache line without any branches and target addresses. The (b) example with a conditional jump in the second word of the cache line assess the first two words as $\frac{1}{2}$ since worst-case is the use of only two out of four words in that cache line. If no jump is performed all instructions should have the local miss-ratio $\frac{1}{4}$, but since both scenarios are possible, only the worst-case miss-ratio is assigned to

[2]A *basic block* is a linear sequence of instructions without halt or possibility of branching except at the end

$$\begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/4 & \\ 1/4 & \\ 1/4 & \\ 1/4 & \end{array}\right. \\ \downarrow \end{array} \qquad \begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/2 & \\ 1/2 & \rightarrow \\ 1/4 & \\ 1/4 & \end{array}\right. \\ \downarrow \end{array} \qquad \begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/4 & \\ 1/4 & \\ 1/2 & \leftarrow \\ 1/2 & \end{array}\right. \\ \downarrow \end{array}$$

(a) (b) (c)

$$\begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/4 & \\ 1/3 & \leftarrow \\ 1/3 & \\ 1/3 & \rightarrow \end{array}\right. \\ \downarrow \end{array} \qquad \begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/4 & \\ 1/4 & \\ 1/2 & \leftarrow \\ 1/1 & \leftarrow \end{array}\right. \\ \downarrow \end{array} \qquad \begin{array}{c} \downarrow \\ \left\{\begin{array}{ll} 1/2 & \\ 1/2 & \rightarrow \\ 1/4 & \\ 1/1 & \leftarrow \end{array}\right. \\ \downarrow \end{array}$$

(d) (e) (f)

Figure 7.1: Some examples of *"local cache miss-ratio"* on 4-word cache lines. Conditional and unconditional jumps are symbolized as ($\rightarrow$) and jump targets as ($\leftarrow$).

a word. In (c) the third word is a target address and since there is a possibility that only two words are being used in this cache line, the two last words are assigned $\frac{1}{2}$ as local miss-ratio.

A way to bound the WCCMR tighter is to distinguish short forward and backward jumps within a cache line. If for instance Figure 7.1(f) would be such a case the miss-ratios would be $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{4}, \frac{1}{3}\}$ and for a short backward loop, as in 7.1(d), all words in the line would be assessed $\frac{1}{4}$.

7.4.4 Loops

If the loop has a higher miss-ratio than the rest of the program it is important to know how large this part of the execution is relative to the rest of the program. By using the *maximum* possible number of iterations in the loop will render a safe estimation if the miss-ratio in the loop is higher than in the rest of the program. If the miss-ratio in the loop is lower than in the rest of the program, the *minimum* possible number of iterations must be used. It is non-trivial to determine which of those that should be used until the complete program is analyzed, and that is why to the best of our knowledge no major pruning or substitution method is possible.

It is therefore necessary to know the number of iterations for the loops in the analyzed program. To determine the number of iterations is trivial for a `for`-loop with a simple counter, assuming the the counter is not changed in the loop body. For loops with more general termination conditions, the number of iterations is not that obvious.

A common method to handle such cases is to add manual annotations to the code (see for example[PK89]). There are however automated methods that can determine the maximum number of iterations without manual annotations (see for example [EG97, Gus00]). Such methods could be used prior to our analysis.

7.4.5 The possible-execution-paths tree

All basic blocks in the CFG associate to a tuple $\langle mr, weight, min, max \rangle$ where *mr* is *miss-ratio, weight* the number of instructions in the basic block, *min* the minimum number of execution times and *max*, the maximum number of execution times of the basic block. Non-iterating sequences of code will have *min* and *max* assigned to '1'.

A binary tree of possible execution scenarios is generated. An if-statement generates two possible branches and loops generate also two branches: the minimum-iteration-path and the maximum-iteration-path.

Some optimizations can be performed to reduce the size of the tree.

- A loop with a fixed number of iterations can be simplified to a basic block following the previous basic block. Such a loop (for instance a `for`-statement) is identified when the minimum number of iterations of a basic block is equal to the maximum.

- If two paths have identical tuple-descriptions, those can be reduced to one path and the branch in the tree can be omitted.

- If it can be proved that the rest of the program (at a certain stage) will not be able to reach a higher WCCMR than another path, the search in that branch can be aborted and the execution path may be omitted in the tree. Such a proof is possible to make if the number of executable instructions are so few that the already calculated part of the program will outweigh the rest even if the remaining program will have a miss-ratio of 100%.

This information can be available through a generation of an execution path tree that is built reversal from the end and built upon the number of instructions executed in each basic block. This “weight tree” must not be complete, but the closer it is reaching the start of the program the more optimizations can be performed at the possible-execution-path-tree. The “weight tree” can easily be optimized since only the heaviest node of all duplicated nodes are necessary — all other nodes and paths to light weight nodes can be omitted.

These optimizations can be performed directly after the loop analysis, before the tree generation, to simplify the tree generation process.

7.4.6 The overall miss-ratio

The last step is to compute the WCCMR in all possible execution paths to find its maximum value. Every node is only once traversed but observe that the complete path must be traversed to calculate a correct value of WCCMR.

7.4.7 Algorithm performance

The performance consuming parts of the algorithm are focused on two parts. The first is the construction of the CFG and assign all assembly instructions with a local miss-ratio, which can be performed with the efficiency of O(n) where n is the number of assembly instructions in the program.

The second part is the traversing of the CFG to build and analyze the tree, which is built with combinations of basic blocks in all execution paths. Since each node in the CFG can occur multiple times in the tree, there is a possibility of an exponential growth of the tree O(2^n), which seems to be the algorithms bottleneck. Even a small program but with a complex structure of loops and branches can take very long time to analyze. The situation is however not as bad as it might first appear since only a fraction of all execution paths are analyzed; In loops only the minimum and maximum number of iterations are considered, not the complete interval, and several branch situations can be omitted by optimization.

7.5 Example

The code in Figure 7.2 is transformed to the CFG in Figure 7.3.

A straight-line edge in the figure represents a basic block and a curved edge represents a conditional or unconditional jump in the code. The `do-while` loop is analyzed to iterate anything between 4-15 times and the `while` loop will iterate 1 or 2 times.

```
...
...
if(a>b) {
  ...
  ...
  do{
    ...
  }while(c>d);
}
else {
  ...
  ...
  while(e<3){   .
    ...
  }
}
...
```

Figure 7.2: A code written in C that will be analyzed as an illustration of the method.

In this example we have chosen the cache line to be 8 words as in for instance in Motorola PowerPC 750. A CFG is constructed and from this each instruction is assigned a local cache miss-ratio (Figure 7.3). The program is analyzed to determine each loop's maximum and minimum number of iterations.

The binary tree in Figure 7.4 describes all possible execution paths to terminate the program. The `if-else`-statement leads to two branches (true/false) and the loops will also generate two branches each (maximum and minimum number of iterations). Each edge in the tree symbolizes a basic block and it's associated cache miss-ratio weight.

Table 7.1 shows the calculated cache miss-ratio for the different execution paths. The execution time calculation should just give a hint about what is going on; we let each miss gives ten “time units” penalty.

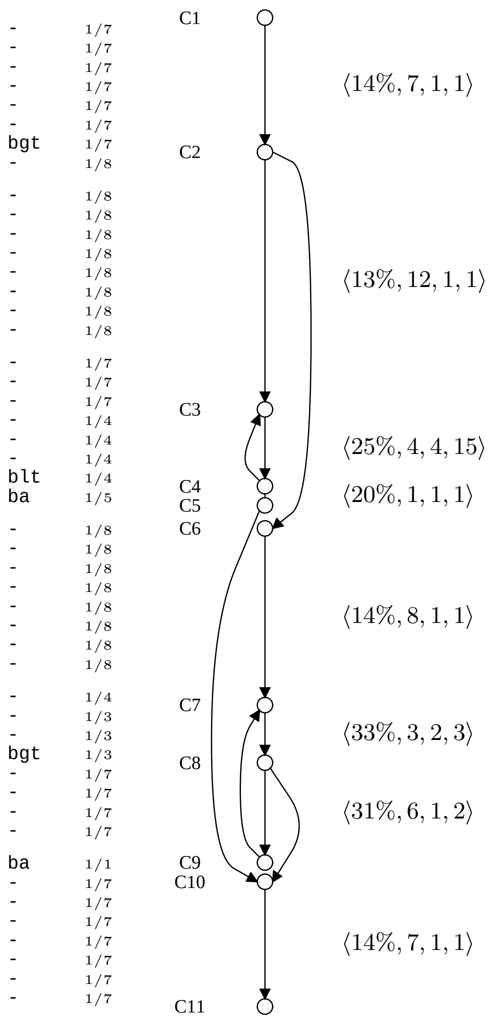

Figure 7.3: The core of the proposed method. The column to the very left is the *assembly language* of the C-code in Figure 7.2. The next column describes each assembly instructions *"local miss-ratio"* that is derived from the *control flow graph* (CFG) in the middle. The last column to the right describes each *basic block's miss-ratio and its potential share weight* of the complete program.

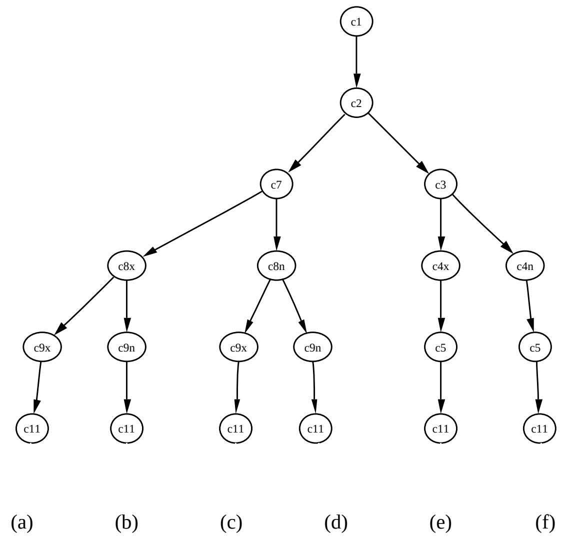

Figure 7.4: The binary search tree to find the WCCMR. The node labels corresponds to the node labels in Figure 7.3. The suffix 'n' and 'x' in some nodes indicates the path of the minimum and maximum number of iterations in a loop.

As shown in Table 7.1, execution path (e) will yield the highest cache miss-ratio. One can notify that path (c) has a shorter execution time than (f), but a higher miss-ratio, and the conclusion from this is that only a combination of many instructions and a high miss-ratio will render a long execution time. An other view is that the worst-case execution time path must not be the same as the worst-case cache miss-ratio path. Observe that the path (e) doesn't contain the basic blocks with the worst miss-ratio, and the reason is that those blocks weighted little because of the complete programs execution behavior.

7.6 Future work

A reduction method of the tree to keep it manageable and reduce the analysis time will be developed. It might however not be possible without approxima-

Path	Miss-ratio	Number of instr.	Exec. time
(a)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 3 + 31 \cdot 6 \cdot 2 + 14 \cdot 7)/43 = 20.6\%$	43	132
(b)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 3 + 31 \cdot 6 \cdot 1 + 14 \cdot 7)/37 = 18.9\%$	37	107
(c)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 2 + 31 \cdot 6 \cdot 2 + 14 \cdot 7)/40 = 19.7\%$	40	119
(d)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 2 + 31 \cdot 6 \cdot 1 + 14 \cdot 7)/34 = 17.6\%$	34	94
(e)	$(14 \cdot 7 + 13 \cdot 12 + 25 \cdot 4 \cdot 15 + 20 \cdot 1 + 14 \cdot 7)/87 = \mathbf{21.5}\%$	87	275
(f)	$(14 \cdot 7 + 13 \cdot 12 + 25 \cdot 4 \cdot 4 + 20 \cdot 1 + 14 \cdot 7)/43 = 18.0\%$	43	121

Table 7.1: Calculated WCCMR for each of the execution paths of the binary tree in Figure 7.4.

tions with a looser bound of WCCMR as a result.

The method will also be developed to handle temporal locality. This can be achieved by including a static cache simulator as for instance in [AMWH94]. With this extension a tighter bound of the WCCMR can be accomplished, but to the price of a more performance intensive analysis.

7.7 Conclusion

The cache miss-ratio is an important property of a program that controls performance, execution time and power consumption among many other properties.

This paper proposes a simple analysis technique to find a worst-case cache miss-ratio execution path in a program. The cache miss-ratio can for instance directly be used to estimate the highest possible power consumption of a program, but also be used as an input for a compiler optimizition. The method is based on the fact that spatial locality is exploited when several instructions in a cache line are executed consecutively. The more instructions that executes without a jump, the lower cache miss-ratio. All instructions can hereby be assigned a ”local miss-ratio” that can be used to compute over-all cache miss-ratio for different execution paths. To cope with the problem that parts of the program may be more used than others in for instance loops, a method based on abstract interpretation computes the minimum and maximum number of iterations. The method needs no manual annotations and can be fully automated.

This paper also demonstrates with an example that the worst-case execution time path must not be the same as the worst-case cache miss-ratio path.

The major drawback of the proposed method at this stage is the exponential growth of the search tree that demands high performance to solve complex program structures in a reasonable time. The method as presented is suitable for programs that is partitioned into many, small processes[3]. Indirect pre-emption effects must not be concerned since temporal locality is not included in the method and will by this not suffer from cache-related pre-emption delay and hereby still yield safe values.

[3]also referred as *tasks* in real-time systems

Paper D

Filip Sebek

When does a disabled instruction cache out-perform an enabled?

Submitted to Design, Automation and Test in Europe (DATE) 2003

München, Germany.

Chapter 8

When does a disabled instruction cache out-perform an enabled?

Abstract

This paper presents a method to determine at what cache miss-ratio level a disabled cache memory gives better system performance than an enabled. The approach is to measure the execution time on synthetic workbench programs with fixed cache miss-ratio levels. A direct measurement instead of a simulation or static analysis is simpler and faster to perform since computer systems often are very complex with advanced CPUs, busses and several layers of cache memories.

Experimental measurement results performed on an MPC750-based computer system showed that this level is so high (84%) that is almost impossible to reach for instruction caching. This indicates that cache memories can be enabled in virtually any real-time system without breaking timing constraints and hazarding the functionality.

8.1 Introduction and motivation

8.1.1 Caches and real-time

Cache memories are today common in computer systems to bridge the response time from primary memory to boost up performance. As a consequence it also reduces bus traffic and may also reduce power consumption in the system. Since the small cache is too small to hold all data and instructions, blocks are swapped in and out depending on what section of code in the program that is handled at the moment.

In real-time systems time constraints must be satisfied to guarantee correct function. Since cache memories will make instruction and data fetch time variable, execution time will be very complicated to calculate. The miss penalty is much higher than a single primary memory access, so there is a possibility that the execution time may even be longer for parts or even the complete program compared to a system without caches[1]. Due to the cache memories highly complex behavior, they are very often disabled in hard real-time systems to make them safe. This very pessimistic real-time approach of the always cache missing program will in this paper be unveiled.

8.1.2 Caches and locality

To understand the proposed method it is essential to understand how modern CPUs and cache memories work in a computer system.

One fundament of cache memories is *locality* that either can be temporal or spatial.

- *Temporal locality* (also called *locality in time*) concerns time. If a program is accessing an address the chance is higher that the same address is used in the near future, compared to an arbitrary other address.

- *Spatial locality* (also called *locality in space*) states that items that are close to each other in address space tend to be referred to close in time too.

The statements above builds on the fact that instructions are formed in sequences and loops, and that data is often allocated as stacks, strings, vectors and matrices. With the spatial behavior of programs in mind, much sense would be to load more data from the underlying hierarchic memory at once

[1]In this paper absence of cache memories and to disable a cache will be considered as the same

to take advantage of greater bandwidth and reduce the penalty of long latency that characterizes primary memory. This chunk of data is called *cache block*[2] and is the smallest piece of data a cache handles.

If the requested data isn't in the cache, it is called a *miss* and a cache block must be fetched from the primary memory. Since miss-time can be multiple times longer than hit-time and miss-ratio is commonly very low, a miss is considered as a *penalty*.

Since cache memories are so common today, primary memory has been developed to send a burst of data to fill up the complete block when a primary memory access is requested. The burst contains a sequence of consecutive data in the memory and the main advantage of burst mode is that the overhead[3] of a memory access can be decreased.

8.1.3 An overview of the method

The procedure to determine the threshold level of the miss-ratio is roughly as follows:

1. Generate several tasks with different fixed cache miss-ratio.
2. Run tasks with enabled instruction cache and measure the execution time for each task.
3. Run a task with disabled cache and measure the execution time.
4. Calculate execution time for a single instruction ($\frac{execution\ time}{\#instructions}$)
5. Interpolate and compare

8.2 Generation of code with a fix cache miss-ratio

The generation of code with a fix miss-ratio has been accomplished in two ways. Either spatial locality is exploited by only executing a fix fraction of the instructions in a cache block or a fraction of reusable code is used to exploit temporal locality. A third possibility is to combine both these methods.

a) If the first instruction in all cache blocks is an unconditional jump to the start of the next cache block only the first word in each block will be

[2]A cache *block* is sometimes also called *line*

[3]The overhead is in this case RAM search, bus arbitration and handshaking protocols.

```
{ L0 :  jump L1    (m)        { L0 :  nop        (m)
{       not used              {       jump L1    (h)
{       not used              {       not used
{       not used              {       not used
{ L1 :  jump L2    (m)        { L1 :  nop        (m)
{       not used              {       jump L2    (h)
{       not used              {       not used
{       not used              {       not used

           (i)                           (ii)

{ L0 :  nop        (m)        { L0 :  jump L1    (m)
{       nop        (h)        {       not used
{       jump L1    (h)        {       not used
{       not used              {       not used
{ L1 :  nop        (m)        { L1 :  nop        (m)
{       nop        (h)        {       jump L2    (h)
{       jump L2    (h)        {       not used
{       not used              {       not used

           (iii)                         (iv)
```

Figure 8.1: Examples of a fix miss-ratio. Each bracket around the code symbolizes a cache block. The misses *(m)* and hits *(h)* elucidate each words cache status.(i) yields 100% misses, (ii) 50%, (iii) 33% and (iv) 66%

accessed and by this never exploit spatial locality. If the active code mass is much larger than the cache memory, the temporal reuse of code will not take place and decrease the miss-ratio. By this we have generated a code with 100% (spatial) cache misses. To generate code with 50% misses the jump is moved from the first to the second word in the cache block, and to generate code with 33% the jump is moved to the third word and so on. 66% misses can be generated by altering the jump from the first and second word in the cache block in the complete program. All these examples are illustrated in Figure 8.1. The information needed to construct this code is the cache size and the block size.

b) Code within a loop that fits into the cache will gain a lower miss-ratio each iteration since it reuses the cache blocks (temporal locality). If the code is a bit larger than the cache size some code will be swapped out and the amount of *useable cache blocks* in the cache will decrease.

$$miss\ ratio = \begin{cases} 0\%, & ts \leq cs \ \wedge \ time \rightarrow \infty; \\ \frac{ts-cs}{cs} \cdot sa, & cs < ts < cs \cdot sa; \\ 100\%, & ts \geq \frac{ts-cs}{cs} \cdot sa; \end{cases}$$

Figure 8.2: The cache miss-ratio of a task that is larger than the cache memory itself depends of the task size and the cache memory's associativity.(*cs* = cache size, *ts* = task size, and *sa* = set associativity

> With code that has 100% spatial misses the cache miss-ratio will be as described in the formula in Figure 2.
>
> Example: A 38kB task with 100% spatial misses will in a 32kB 2-way set-associative cache memory obtain an average miss-ratio of $\frac{38-32}{32} \cdot 2 = 37.5\%$

A more correct description of "task size" is "the task's cache-non-interfering *active* or *useful* cache blocks" but in this synthetic generated workload these terms turn to be equivalent since all blocks in the task are active. Besides cache and block size also knowledge about associativity is needed. Observe that the second method is useable on LRU and FIFO exchange algorithms only — never random.

8.3 Experimental results

8.3.1 The target system

Observe that none of the following described special hardware is necessary since the only property that is measured is execution time. The only thing one really need is some software where the number of executed instructions and the cache miss-ratio is known, and a stopwatch. The advantage with the used system is however that the number of executed instructions and the cache miss-ratio can be measured (instead of estimated). The system will in this paper only be described briefly with focus on some highlights of important issues in this very context, and for those who are interested in details are directed to read [Seb02a].

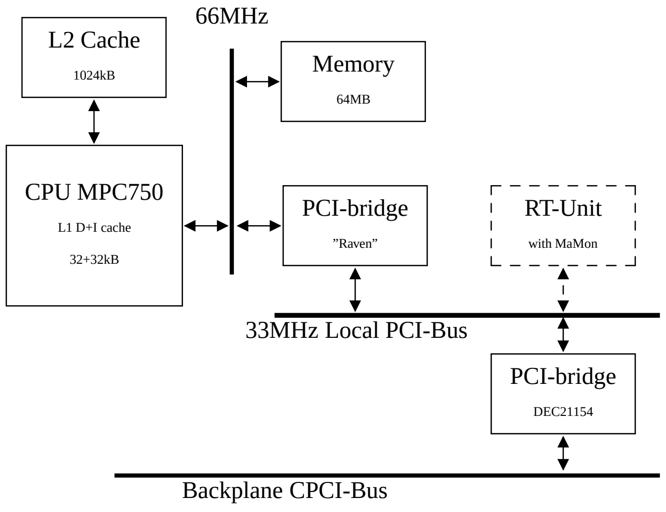

Figure 8.3: CPU-card. The RT-Unit is only on master cards.

8.3.1.1 SARA overview

SARA — Scaleable Architecture for Real-Time Applications — is a research project with a Motorola CPX2000 Compact PCI backplane bus with at most eight Motorola Power PC750-processor (MPC750) boards [LKF99]. The processor boards are equipped with a MPC750 running at 367MHz that is connected to a 66MHz bus as well as the 64MB DRAM main memory (Figure 8.3).

8.3.1.2 MPC750

The microprocessor MPC750 has a split instruction and data cache at the first level, and a unified cache memory at the second level. The first level caches are 32kB each and organized in 8-way set-association with a pseudo LRU replacement policy. Each cache block can hold eight 32-bit words. The second level cache is in the target system 1024kB, 2-way set-associative and the block size is in this case 128 byte that is divided into sub-blocks. The second level cache and the data cache are disabled in the described experiment to not pollute the results.

The MPC750's is equipped with an on-chip performance monitor that can monitor 48 different kinds of events, but there are only 4 performance monitor counters (PMCs) available. Only 5 of the 48 events can be associated with any PMC and the rest are associated to a dedicated PMC [Mot97, Mot99].

8.3.1.3 Real-Time Unit

A special *master card* is equipped with a Real Time Unit (RTU)[FSLA95] that controls the execution of the tasks on all processor cards. The RTU is a high performance and performance predictable hardware implementation of an operating system kernel that handles scheduling and other real-time operating system services. The other processor cards are used as *slaves* to increase application performance.

8.3.1.4 MAMon — an application monitor

A special device called *Multipurpose Application Monitor (MAMon)* [ES01, ES02], can tap for instance the RTU non-intrusively on information regarding context-switching, inter process communication, task synchronization etc. There is also a possibility to write to special registers called *software probes* directly from the application.

After the data is collected and time stamped with a high resolution clock, it is continuously sent through a parallel port to an external database host for post analysis.

Today MAMon and the RTU co-exists in the same FPGA, and besides increased performance this is a very practical and cost effective way to eliminate problems with PCB-layout and other hardware manufacturing issues.

8.3.2 Workbench

Finding the threshold value where a system with a disabled cache memory out-performs an enabled mustn't be done with trial-and-error.

The average execution time of each instruction is easily calculated by divide the time it takes to execute a task by the number of executed instructions. It is this quota that is to be compared. Since the instruction's execution time grows linearly with increased miss-ratio, the average execution time can be interpolated between two measurements.

Theoretically only three measurements are necessary to perform to get a threshold miss-ratio; two with enabled cache and one with disabled, but since all measurement includes some divergence, it is safer to measure at more miss-ratio levels.

A ”dummy” MAMon software probe has been inserted in each workbench task. The contents of the probe is of no interest here, but since all probe events are time stamped it is used to measure the time it takes to execute all instructions in the task.

8.3.3 Results

Five tasks with different fixed miss-ratios were generated by using large tasks with chosen jump-instructions in the cache block as the type (a) described in section 8.2. The miss-ratio and the numbers of executed instructions were measured through the MAMon tool to verify the correctness of the generated code. Then a task was executed but with disabled caches to have something to compare with. The instructions' average execution time were calculated and plotted into a graph. All results are presented in Figure 8.4.

8.3.4 Discussion

A program without jumps backwards will never take advantage of temporal locality and one can then assume that the miss-ratio would be quite high since the key concept of caching isn't applied. One shouldn't at this stage come to the conclusion that this will yield a miss-ratio of 100% since caches use blocks that are filled up in bursts and will by this exploit spatial locality. Best case is when all the words in the cache block is used with a miss-ratio of $\frac{1}{cache\ block\ size}$. To get 100% misses only one word in the cache block must be used which can be achieved by either jump from the first word in the block to another cache block or jump to the last word in the block. To achieve a miss-ratio of 50% in a system with 8-word cache blocks the code must contain something between 12,5%–50% jump-instructions. A program where every second word is a jump doesn't do anything more than jumping (since the other half examines the conditions and there is no space left for “real” computation) and such programs don't exist.

Cache blocks with 8 words or more, 32-bit data-buses and primary memory that fills cache blocks in bursts mode are not science fiction but reality today. With these facts, instruction caches shouldn't be able to cause missed deadlines in real-time systems. By determining the worst-case miss-ratio of a program

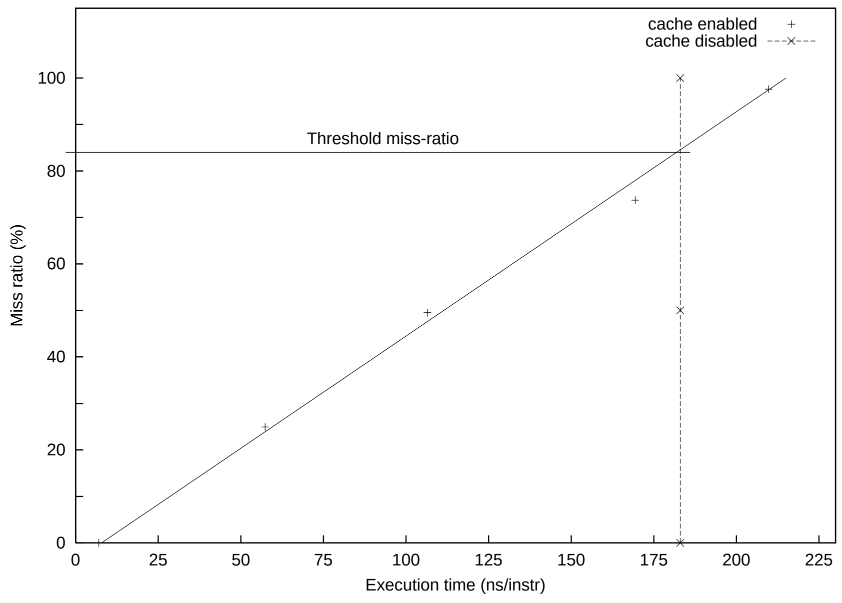

Figure 8.4: The measured performance when the experimental system runs with disabled and enabled instruction caches. An application with an instruction cache miss-ratio that exceeds 84%, should execute with disabled caches.

[SG02], one can be really safe and even utilize the extra performance the cache memory gives.

8.4 Related work

To the best of our knowledge nobody has presented a simple method to find miss-ratio threshold values. There is however very much work done close and related to the presented, especially in the computer architecture and real-time system field, that could be used instead if they were modified. For computer architecture researchers the results of this paper may be less interesting, but for many in the real-time community it is news that can show what areas that are pure academic and not that interesting for applied or industrial research.

In the real-time area two kind of execution time analysis that includes caches are performed; the *extrinsic* cache behavior that analyze how tasks, threads and processes interfere each others cache context and by this cause extra misses and swaps during pre-emption (called *cache-related pre-emption delay — CRPD* or *cache refill penalty*)[AHH89]. The other type analyze the *intrinsic* cache behavior and is more close to the suggested method since it handles the task's or program's internal interference (collision and capacity cache block swaps). After that analysis, the CRPD is then added to the execution time if the system is pre-emptive[BN94] or eliminated with cache partitioning to the price of decreased over-all performance[Kir89, Wol93, Mue95]. Both types of analysis can either be performed statically before execution[HAM$^+$99, TD00, FMWA99, LHM$^+$97] or simulated[MB91, SL88], or measured directly on the target system[LHM$^+$97, PF99].

As motivated, an accurate model of the analyzed system to statically analyze or simulate on may be hard to accomplish[AKP01, Eng01]. The real measurement alternative may in many cases be faster and more accurate even if the observability decreases. It is also easier to perform since the analysist mustn't be an expert on computer architecture, which is the case if a model or simulator must be constructed from scratch.

8.5 Conclusion

In hard real-time systems it is common to disable the cache since the execution time of each instruction is variable which makes the software complicated to analyze. There is an unacceptable possibility that a high cache miss penalty combined with a high miss-ratio might cause a missed deadline and may then jeopardize the safety of the controlled process. A system with disabled caches will however only use a fraction of the available CPU performance, which is a waste of resources.

One way to guarantee that the system functions properly is to use any of the WCET[4]-tools that includes cache memories. The proposed method is however in many cases much faster and easier to use if the system is very complex and no model is available. The method determines the threshold value of cache miss-ratio where a system with disabled cache outperforms an enabled cache. The paper describes how to generate tasks with a fixed miss-ratio and how to use them in the measurement process.

[4]Worst Case Execution Time

Experimental results showed that the threshold miss-ratio was as high as 84% on a modern high-performance computer system.

It is common for modern CPUs to fill a cache block in burst mode from the primary memory. That is why the miss-penalty is relatively low and this is also one reason why a relatively high cache miss-ratio must be achieved to out-perform a system without a cache memory. Such high cache miss-ratio is almost impossible to reach with regular software that runs on a system with primary memories that supports burst mode. To claim that an instruction cache memory may cause a missed deadline in a modern computer real-time system is nothing but an academic myth.

Bibliography

[AHH89] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Theory of Computing Systems*, 7(2):184–215, May 1989.

[AKP01] Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, December 3, 2001.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, 1995.

[AMWH94] Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the IEEE Real-Time Systems Symposium 1994*, pages 172–181, December 1994.

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

[BMWS$^+$96] J. V. Busquets-Mataix, A. Wellings, J. J. Serrano, R. Ors, and P. Gil. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[BN94] Swagato Basumallick and Kelvin D. 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.

[CS91] Bryce Cogswell and Zary Segall. Macs – a predictable architecture for real time systems, 1991.

[DNVG01] Paolo D'Alberto, Alexandru Nicolau, Alexander Veidenbaum, and Rajesh Gupta. Static analysis of parameterized loop nests for energy efficient use of data caches. In *Proceedings of Workshop on Compilers and Operating Systems for Low Power (COLP)*, Barcelona, Spain, September 2001.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *EUROPAR97*, pages 1298–1307, August 1997.

[Eng01] Jakob Engblom. On hardware and hardware models for embedded real-time systems. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, December 3, 2001.

[ES01] Mohammed El Shobaki. Non-intrusive hardware/software monitoring for single- and multiprocessor real-time systems. Technical report, Mälardalen Real-Time Research Centre, Västerås, Sweden, April 2001.

[ES02] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2–3):163–189, November 1999.

[FSLA95] Johan Furunäs, Johan Stärner, Lennart Lindh, and Joakim Adomat. RTU94 – real time unit 1994 – reference manual. Technical report, Dept. of computer engineering, Mälardalen University, Västerås, Sweden, January 1995.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. Doctorial thesis, Uppsala University and Mälardalen University, Västerås, Sweden, May 2000.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[Kar90] Rabindra P. Kar. Implementing the rhealstone real-time benchmark. *Dr. Dobb's Journal*, pages 46–55 and 100–104, April 1990.

[Kir88] David B. Kirk. Process dependent static cache partitioning for real-time systems. In *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, pages 181–190. IEEE Computer Society Press, 1988.

[Kir89] David B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1989*, pages 229–239, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.

[KMH96] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 230–240, Brookline, Massachusetts, USA, June 10–12, 1996.

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995. Best Papers of the Real-Time Systems Symposium, 1994.

[LHM+97] Chang-Gun Lee, Joosun Hahn, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 18th Real-Time System Symposium*, pages 187–198, San Francisco, USA, December 3–5, 1997. IEEE Computer Society Press.

[LHS$^+$96] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 17th Real-Time System Symposium*, pages 264–274, December 1996.

[LKF99] Lennart Lindh, Tommy Klevin, and Johan Furunäs. Scaleable architecture for real-time applications – SARA. In *Proceedings of SNART 1999*, Linköping, Sweden, 1999.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLL$^+$98] Sheayun Lee, Chang-Gun Lee, Minsuk Lee, Sang Lyul Min, and Chong Sang Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 51–64, June 1998.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation (DAC'95)*, page 6, San Francisco, CA, USA, June 12–16, 1995.

[LML$^+$94] Sung-Soo Lim, Sang Lyul Min, Minsuk Lee, Chang Park, Heonshik Shin, and Chong Sang Kim. An accurate instruction cache analysis technique for real-time systems. In *Proceedings of the Workshop on Architectures for Real-time Applications*, April 1994.

[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 Real Time System Symposium*, pages 298–307, December 1995.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

[LS99a] Thomas Lundqvist and Per Stenström. Emperical bounds on data caching in high-performance real-time systems. Technical Report 99-4, Department of Computer Engineering, Chalmers, Göteborg, Sweden, April 1999.

[LS99b] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, pages 183–207, November 1999. Special Issue on Timing Validation.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, USA, April 1991.

[Mot97] Motorola Corp. *MPC750 RISC Microprocessor Users Manual*, August 1997.

[Mot99] Motorola Corp. *Errata to MPC750 RISC Microprocessor Users Manual*, July 1999.

[Mue95] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, USA, June 1995.

[NL00] Peter Nygren and Lennart Lindh. Virtual communication bus with hardware and software tasks in real-time system. In *Proceedings for the work in progress and industrial experience sessions at 12th Euromicro conferance on Real-time systems*, June 2000.

[NR95] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *ACM SIGPLAN Notices*, 30(11):20–30, 1995.

[OS97] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[PF99] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th Real-Time Computing Systems and Applications RTCSA*, Hong-Kong, December 13–15, 1999. IEEE Computer Society.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[Raw93] Jai Rawat. Static analysis of cache performance for real-time programming. Master thesis TR93-19, Iowa State University of Science and Technology, November 17, 1993.

[SA97] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. Technical Report 27/97, C-Lab, Paderborn, Germany, December 9 1997.

[Seb01] Filip Sebek. Cache memories in real-time systems. Technical Report 01/37, Mälardalen Real-Time Research Centre, Department of Computer Engineering, Mälardalen University, Sweden, September 29, 2001.

[Seb02a] Filip Sebek. The real cost of task pre-emptions — measuring real-time-related cache performance with a hw/sw hybrid technique (paper b). Technical Report 02/58, Mälardalen Real-Time Research Centre, August 2002.

[Seb02b] Filip Sebek. When does a disabled instruction cache outperform an enabled? (paper d). In *Submitted to DATE03*, München, Germany, March 2002.

[SG02] Filip Sebek and Jan Gustafsson. Determining the worst-case instruction cache miss-ratio (paper c). In *Proceedings of ESCODES 2002*, San José, CA, USA, September 2002.

[SL88] Robert T. Short and Henry M. Levy. A Simulation Study of Two-Level Caches. 1988.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Stä98] Johan Stärner. Controlling cache behavior to improve predictability in real-time systems. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[TD00] Hiroyuki Tomiyama and Nikil Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES 2000)*, pages 67–71, May 2000.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Real-Time Systems*, 18(2/3), May 2000.

[Tha00] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. Doctorial thesis, Royal Institute of Technology, KTH and Mälardalen University, Stockholm and Västerås, Sweden, May 2000.

[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 Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 192–202, Washington - Brussels - Tokyo, June 1997. IEEE.

[WMH$^+$99] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2–3):209–233, 1999.

[Wol93] Andrew Wolfe. Software-based cache partitioning for real time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, September 1993.

Index

Motvind lär dig segla
— Loesje

Previous thesis at MdH