2012

WCET Optimizations and Architectural Support for Hard Real-Time Systems

Yiqiang Ding
Virginia Commonwealth University

WCET Optimizations and Architectural Support for Hard Real-Time Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

Yiqiang Ding
B.S., Beijing University of Posts and Telecommunications, China, 2002
M.S. Beijing University of Posts and Telecommunications, China, 2005

Director: Dr. Wei Zhang,
Associate Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
December, 2012

ACKNOWLEDGMENTS

I would like to thank Dr. Wei Zhang for his invaluable assistance and insights leading to my Phd research and this dissertation in the last five years.

My sincere thanks also goes to the members of my graduate committee for their patience and understanding during the numerous time of effort that went into the production of this dissertation.

Last but not the least, many thanks go to my wife and my parents from the bottom of my heart.

Abstract

WCET ANALYSIS AND OPTIMIZATIONS OF THE REAL-TIME

APPLICATIONS ON MULTI-CORE PROCESSORS

By Yiqiang Ding, Ph.D.

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2012.

Director: Dr. Wei Zhang,
Associate Professor, Department of Electrical and Computer Engineering

As time predictability is critical to hard real-time systems, it is not only necessary to accurately estimate the worst-case execution time (WCET) of the real-time tasks but also desirable to improve either the WCET of the tasks or time predictability of the system, because the real-time tasks with lower WCETs are easy to schedule and more likely to meat their deadlines. As a real-time system is an integration of software and hardware, the optimization can be achieved through two ways: software optimization and time-predictable architectural support.

In terms of software optimization, we first propose a loop-based instruction prefetching approach to further improve the WCET comparing with simple prefetching techniques such as Next-N-Line prefetching which can enhance both the average-case performance and the worst-case performance. Our prefetching approach can exploit the program control-flow information to intelligently prefetch instructions that are most likely needed. Second, as inter-thread interferences in

shared caches can significantly affect the WCET of real-time tasks running on multicore processors, we study three multicore-aware code positioning methods to reduce the inter-core L2 cache interferences between co-running real-time threads. One strategy focuses on decreasing the longest WCET among the co-running threads, and two other methods aim at achieving fairness in terms of the amount or percentage of WCET reduction among co-running threads.

In the aspect of time-predictable architectural support, we introduce the concept of architectural time predictability (ATP) to separate timing uncertainty concerns caused by hardware from software, which greatly facilitates the advancement of time-predictable processor design. We also propose a metric called Architectural Time-predictability Factor (ATF) to measure architectural time predictability quantitatively.

Furthermore, while cache memories can generally improve average-case performance, they are harmful to time predictability and thus are not desirable for hard real-time and safety-critical systems. In contrast, Scratch-Pad Memories (SPMs) are time predictable, but they may lead to inferior performance. Guided by ATF, we propose and evaluate a variety of hybrid on-chip memory architectures to combine both caches and SPMs intelligently to achieve good time predictability and high performance.

Detailed implementation and experimental results discussion are presented in this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Real-time systems are widely used in our society such as automobile and aircraft controllers. Besides performance, time predictability is also critical to real-time systems, especially hard real-time systems. Missing deadlines in those systems may either lead to catastrophic consequences or decrease quality of services. The Worst-Case Execution Time (WCET) of an application must be calculated to determine if its deadline can be always met. It is desirable not only to accurately estimate the WCET, but also to optimize it, because the reduction of the WCET of the real-time tasks can improve the feasibility of the scheduling of those tasks. Also the improvement of the WCET can conserve the power consumption of the processors, because one can determine the worst-case number of cycles required for a task and lower the clock rate to still meet the deadline with less slacks.

There are two main factors that determine the WCET of a program: first the possible flows of instructions of a program, second the time needed for each instruction in each possible flow [1]. Both factors do not only determine the WCET of the program, but also the complexity of WCET analysis. Possible flows of instructions depend on both the algorithm used to implement the program and the code compilation (software). The time of the execution of each instruction depends on the features and the configurations of the processors (hardware) on which the instructions are executed. Therefore it is possible and necessary to

perform WCET optimizations through either software optimization techniques or architectural support.

Due to the prohibitive cost of worst-case timing analysis for modern processors, especially multicore processors, the design of time-predictable processors has become increasingly important for hard real-time and safety-critical systems. On the other hand, designing a microprocessor with high time predictability but low performance is likely to be useless. However, to the best of our knowledge, currently there is no effective and widely accepted metric to quantitatively evaluate time predictability of processors, which greatly impedes the advancement of time-predictable processor design.

Scratch-Pad Memory (SPM) is an alternative on-chip memory to the cache, which has been increasingly used in embedded processors due to its energy and area efficiency. In a processor with SPM, the mapping of program and data elements into the SPM can be performed either by the user or the compiler, resulting in statically predictable memory access time. However, the performance of SPMs is generally not as good as that of caches because caches can dynamically reuse their space efficiently to benefit more instructions and data. Processors that employ caches or SPMs alone can only benefit either the average-case performance or the time predictability,not both. Guided by a quantitative metric of time predictability of the microprocessor, it is possible and desirable to exploit the hybrid on-chip memory architecture to achieve both time predictability and high performance.

Motivated by these challenges, the rest of this dissertation is organized as follows.

Chapter 1 provides the background knowledge.

Chapter 2 and Chapter 3 discusses two WCET-oriented software optimization techniques respectively. First, Chapter 2 studies a compiler-directed instruction prefetching technique to overcome the deficiencies of the Next- N-Line prefetching [9, 10]. Specifically, we find that since many real-time applications are loop-intensive, the instruction cache pollution due to the Next-N-Line prefetching can frequently occur at loop boundaries, which can adequately affect the performance. To solve this problem, we propose to modify the Next-N-Line prefetcher by prefetching instructions from the beginning of the loop, instead of the subsequent instructions after the loop, when the loop branch (i.e., the back-edge branch [20]) is being executed. We have also discussed the architectural and compiler support for the proposed loop-directed prefetching technique. Our experimental results indicate that the loop-directed prefetching can achieve both better worst-case and average-case performance than the Next-N-Line prefetching.

Chapter 3 studies three approaches — a worst-case-oriented approach (WCO) and two fairness-oriented approaches, including the percentage-fairness-oriented (PFO) and amount-fairness-oriented (AFO) schemes, all of which are based on the WCET analysis on a multi-core processor with a shared L2 cache, but with different optimization goals. Our experiments show that all three proposed techniques can effectively reduce the WCET of co-running

real-time threads to achieve their goals respectively.

In terms of architectural support, Chapter 4 first introduces the concept of timing contract and architectural time predictability (ATP) to separate the timing unpredictability concern caused by hardware design from software, thus making it feasible to quantitatively assess and guide the time-predictable architectural design; Then we propose to use Architectural Time-predictability Factor (ATF) as a metric to quantitatively evaluate architectural time predictability of a processor, as well as architectural time predictability of various architectural and microarchitectural components of the processor. In addition, we evaluate the ATF of a VLIW processor as well as its microarchitectural components, including caches, parallel pipelines, branch predictor, speculative execution and the use of SPM.

Guided by ATF to evaluate the time predictability of a processor, Chapter 5 first proposes hybrid SPM-cache architectures that can leverage SPMs to achieve time predictability while allowing the use of caches for instructions and/or data not stored in the SPMs t o improve the average-case performance. Second, we have systematically explored seven different hybrid on-chip memory architectures to understand how to make best use of both caches and SPMs to store instructions and data for balancing performance and time predictability. Third, while most prior works indicate performance and time predictability generally conflict with each other, this research shows that it is possible to exploit hybrid architectures intelligently for improving both time predictability and performance.

Finally, Chapter 6 concludes this dissertation.

CHAPTER 1

BACKGROUND

In this chapter, background information is provided for the topics covered in this dissertation.

1.1 REAL-TIME SYSTEM

A real-time system is any information processing system which has to respond to externally generated input within a finite and specified deadline. The correctness of the real-time system depends not only on the results it produces, but also on the time it finishes the computation. Such systems play a critical role in modern industrial technologies and safe-critical systems, such as automobile, aircraft, power plant and so on.

The unique characteristics of the real-time systems which are distinguished from the common computing systems are listed as follows:

- Hard deadline: missing a deadline causes a total system failure.
- Soft deadline: missing a deadline degrades the quality of service of the system because the degradation of the usefulness of the result.
- Not fast computing but time-predictable computing: the accurate estimation of Worst-case Execution Time (WCET) of a real-time application is desired.
- Safe-critical applications: missing deadline may result in human lives endanger or catastrophic outcomes

- Embedded systems: real-time systems are usually offered through embedded systems.

The safe and accurate estimation of WCET of the real-time applications is a key requirement of real-time systems. The WCET is defined as the computing upper bounds for the execution times of pieces of code for a given application, where the execution time of a piece of code is defined as the time it takes a processor to execute it. The WCET estimation is demonstrated in Figure 1.1.

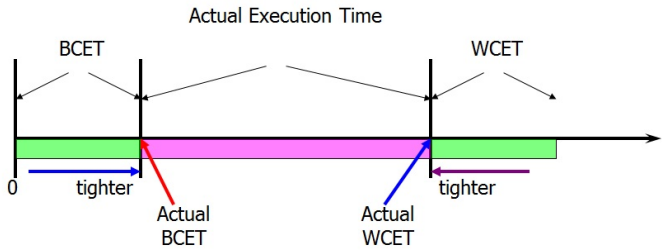


Figure 1.1. The WCET estimation

1.2 MULTICORE PROCESSOR

Multicore processors have already become the mainstream of current server and desktop computer markets. Because of the difficulty to increase the frequency of processors to improve the performance, manufactures intend to integrate multiple processors into a single integrated circuit die. Compared with the single-core processor, the multicore processor can achieve higher performance with lower power consumption. As the next-generation real-time systems need to process exponentially growing volumes of time-sensitive data streams from physical sensors and instruments, the performance boost of mutlicore processors

can support the high performance computing required by the evolution to the next-generation real-time systems.

However, multicore processors also bring challenges to real-time systems. The WCET analysis of multicore processors is much harder than that of single-core processors due to the inter-thread interferences accessing shared resources (e.g. shared bus or cache), which are very difficult to be analyzed statically. For example, the shared L2 cache is an important and widely used design in multicore processors, because it can make multiple cooperative threads to shared instructions/data and the limited on-chip memory efficiently.

1.3 ON-CHIP MEMORY

In order to boost the performance of modern processors, the on-chip memory is used to shorten the gap between the processor speed and memory access time. One type of on-chip memory is cache. It stores the data requested before so that future requests to the data can be served faster. If the data requested are found in the cache, it results in a cache hit, otherwise it refers to a cache miss. The latency of a cache miss is much longer than that of a cache hit, so the cache performance impacts the performance of the processor. Because the data stored in the cache can be replaced by other data dynamically, cache misses always happen due to the dynamic run-time behavior of the processor. Therefore, the cache has unpredictable timing performance which is not desirable in real-time systems

Scratchpad memory (SPM) is an alternative technique of on-chip memory. SPMs are some small physical separate memories directly mapped into the address

space of the main memory system. SPMs can bring some advantages over caches in both performance and energy because of its simple architecture and fast access speed. Furthermore, compare with caches, the timing performance of SPMs is predictable if memory objects are allocated in them statically, because there is no replacement happening to these memory objects. There are already a variety of commercial processors employing scratch-pad memory available in the market such as Motorola MPC500 [2], ARMv6 [3].

CHAPTER 2

LOOP-BASED INSTRUCTION PREFETCHING TO REDUCE THE WORST-CASE EXECUTION TIME

2.1 CHAPTER OVERVIEW

Although one can accurately measure the actual execution time of a given task, WCET estimate based on measurement alone is generally unsafe because it is typically not feasible to exhaust all the possible program paths, especially for applications with complex control flows. As a result, a static analysis technique (i.e., WCET analysis) becomes a promising approach to obtaining the safe and tight upper bound of the execution time for real-time applications. WCET, however, is not only determined by the application itself, but also heavily dependent on the timing information of the underlying hardware processor. Unfortunately, many architectural features of modern microprocessors such as caches, pipelines, dynamic branch prediction, and speculation, are designed for improving the average-case performance, mostly at the cost of the worst-case performance, making it hard to accurately estimate the worst-case execution time [4, 5, 6]. Particularly, instruction caches are widely used in todays microprocessors to bridge the speed discrepancy between the CPU and the memory. Nevertheless, there is no guarantee that in the worst-case, the accesses to an instruction cache will be hits. As a result, the computation time on a processor with an instruction cache is less predictable. Fortunately, prior work on WCET analysis of instruction

caches [7, 8] reveals that the worst-case performance of instruction caches can be reasonably bounded, and actually it is beneficial to employ instruction caches for real-time systems for achieving better performance [7, 8].

To further improve the instruction cache performance, various instruction prefetching techniques [9, 10, 11, 12, 13, 14, 15, 16, 17, 18] can be used. However, most of these prefetching techniques are designed for reducing the average-case instruction cache misses, and their effectiveness on improving the worst-case performance is largely unknown. A recent work [19] has quantitatively studied the impact of a simple yet effective instruction prefetching techniqueNext-N-Line prefetching [9, 10] on the worst-case execution time. While the Next-N-Line prefetching can adequately enhance the average-case performance, it is less effective and inefficient at improving the worst-case performance. The reason is that the Next-N-Line prefetcher will always prefetch the next N cache lines, regardless of the program control flow, which may lead to excessive conflicts between the prefetched instructions and other useful instructions residing in the cache. This cache pollution effect is especially problematic for worst-case timing analysis, since the WCET analyzer has to conservatively estimate the worst-case cache pollution by considering all the possible instructions that may be affected by the prefetched instructions, due to the lack of runtime information. Therefore, unintelligently prefetching useless or excessive instructions may result in worse WCET or more loosely estimated WCET, both of which will add unnecessary pressure to the real-time scheduler. Moreover, prefetching useless instructions will

waste energy dissipation, which is often an important constraint for embedded systems.

This chapter studies a compiler-directed instruction prefetching technique to overcome the deficiencies of the Next- N-Line prefetching [9, 10]. Specifically, we find that since many real-time applications are loop-intensive, the instruction cache pollution due to the Next-N-Line prefetching can frequently occur at loop boundaries, which can adequately affect the performance. To solve this problem, we propose to modify the Next-N-Line prefetcher by prefetching instructions from the beginning of the loop, instead of the subsequent instructions after the loop, when the loop branch (i.e., the back-edge branch [20]) is being executed. We have also discussed the architectural and compiler support for the proposed loop-directed prefetching technique. Our experimental results indicate that the loop-directed prefetching can achieve both better worst-case and average-case performance than the Next-N-Line prefetching.

The rest of the chapter is organized as follows: We present the loop-directed instruction prefetching approach in Section 2.2. The WCET analysis for the loop-directed instruction prefetching is described in Section 3. Section 4 introduces the evaluation methodology and Section 5 gives the experimental results. Finally, we make concluding remarks in Section 6.

2.2 LOOP-BASED INSTRUCTION PREFETCHING

2.2.1 Motivation

While instruction prefetching was originally proposed to improve the average-case instruction cache performance, it may also be useful to enhance the worst-case performance for real-time applications, provided that it can be used in a time-predictable manner. Particularly, in a multiprogramming environment, a WCET analyzer typically has to conservatively assume that all the instruction cache lines are invalidated after context switches. Consequently, a real-time task will suffer from cold misses, which can only be reduced by the instruction prefetching techniques. Moreover, in a pipelined processor, each instruction miss may stall the pipeline for multiple cycles, leading to poor performance. As the processor speed continues to grow faster than the memory speed, time-predictable instruction prefetching will become increasingly important for future real-time systems that demand high performance.

Recent work [19] shows that the Next-N-Line instruction prefetching [9, 10] can benefit both the average-case and the worst-case performance. However, the degree of improvement in the worst-case performance is rather limited. Also, the worst-case instruction cache misses may even become larger (than those without using prefetching) when the prefetching distance is long [19]. These problems are caused by the rigid policy of the Next-N-Line prefetching policy. Precisely, the Next-N-Line prefetcher will always prefetch the next N cache lines, no matter these instructions are needed or not. While this policy is useful when the program is

executed sequentially, it becomes unhelpful or even problematic when the direction of the prefetching is wrong due to the change of control flows at the execution time.

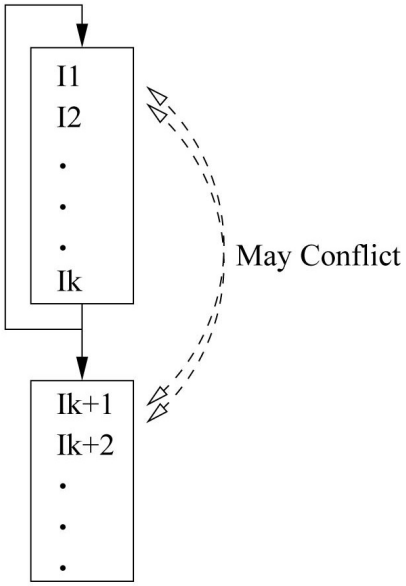


Figure 2.1. A motivation example to illustrate the deficiency of the Next-N-Line prefetching.

For instance, Figure 2.1 shows the control-flow graph of a code segment, which consists of a loop and a basic block. The last instruction of the loop I_k is a back-edge branch, which is likely to be taken for many times as long as the loop iterates, except for the last loop iteration. However, every time when I_k is being executed, the Next-N-Line prefetcher will always prefetch the next N cache lines after I_k, for instance instructions I_{k+1}, I_{k+2}, etc. These prefetched instructions, however, may be mapped to the same cache lines as other instructions within the loop, such as I1, I2, etc., and thus may pollute the instruction cache and degrade the performance. Besides, those prefetched instructions outside the loop will never be executed during the loop execution (except after the last loop iteration), which

should be avoided for both performance and energy reasons.

Since real-time applications are typically loop-intensive, the unintelligent prefetching by strictly following the next N lines policy may significantly reduce the opportunity to optimally benefit performance (as well as energy dissipation). Table 2.1 gives the percentage of execution cycles spent in loops for selected benchmarks (details about the evaluation methodology can be found in Section 2.4). For most of the benchmarks, we find that loop instructions dominate the execution time. On average, 83.9 percent of the total execution time is spent in loops. Therefore, it is important to study an approach to overcoming the deficiency of the Next-N-Line prefetching in order to improve the benefits of instruction cache prefetching on real-time applications.

Table 2.1. The Percentage of Execution Cycles Spent in Loops

Benchmark	# of Loops (%)	Loop Cycles (%)
Bmm	14	96.96%
Fib_mem	1	47.38%
Nested	4	90.55%
Fibcall	1	65.78%
Ludcmp	11	84.91%
Matmul	5	88.07%
Cordic	1	98.62%
Rawcaudio	3	99.27%

2.2.2 The Proposed Approach

To address the above mentioned problem of the Next-N-Line prefetcher [9, 10], we propose a loop-based instruction prefetching technique by intelligently exploiting the program control-flow information that is available at the compilation time. The idea of this approach is that normally instructions can be prefetched sequentially just like what the Next-N-Line prefetcher does; however, when a loop branch is encountered, the instructions in the beginning of the loop (not after the loop) will be prefetched. The reason is that the loop branch is most likely taken. Therefore, by prefetching instructions from the beginning of the loop rather than sequential instructions outside the loop, the direction of the instruction prefetching is kept consistent with the runtime instruction flow (except for the last loop iteration when the loop branch falls through), potentially leading to better performance.

The loop-based instruction prefetching can leverage the existing Next-N-Line prefetcher [9, 10], and its architectural and compiler support can be kept simple and cost-efficient. The hardware support for the loop-directed instruction prefetching is depicted in Figure 2.2. We extend the traditional Next-N-Line prefetcher by adding several components, including a *loop branch address* register, a control signal *LoopBranchEnable*, a hardware table, and a multiplexer. The hardware table is used to store the address of each loop branch and the associated loop header (i.e., the first instruction of each loop). Since both loop branches and loop headers can be identified statically at the compilation time [20], we propose

to store those addresses into the hardware table before we run the program. [1] Typically, the number of loops (or static loop instructions) in a real-time program is relatively small. For instance, as can be seen from Table 2.1, the maximal number of loops in the selected benchmarks is only 14, although they could dominate the dynamic execution cycles. Therefore, the loop address hardware table can be kept small, and hence, will not significantly increase the hardware cost. [2] Also, a small-sized hardware table can make it very time-efficient to perform the associative search for locating a particular entry.

We propose to use the compiler to detect and annotate the loop branches. This can be achieved by using special opcodes for loop branches or exploiting unused fields in the branch instructions. At runtime, when a loop branch

[1] It should be noted that it is possible to update the table at runtime, which however, needs to annotate more instructions in the program and requires more hardware support.

[2] Note that for large applications with many loops, if the hardware table is too small to hold all the prefetch information of the code, we propose to let the compiler place the most frequently used loop information (through static analysis or profiling) into the limited loop table. For the rest of loops whose starting addresses can not be stored in the hardware table, the corresponding loop branches can be annotated to disable the next-N-line prefetcher. Therefore, upon the execution of these annotated loop branches, no instruction will be prefetched, which will not pollute the cache. If the loop branch is taken, the first instruction in the loop body will be executed again, which will enable the next-N-line prefetcher to prefetch instructions in the right direction. In the last loop iteration, the loop branch will not be taken; however, when the first instruction outside of the loop body is executed, the next-N-line prefetcher will also be enabled to start the sequential prefetching, which is also on the right path. Therefore, the impact on the performance is insignificant, even if the table cannot hold the prefetching information of all loops in large applications.

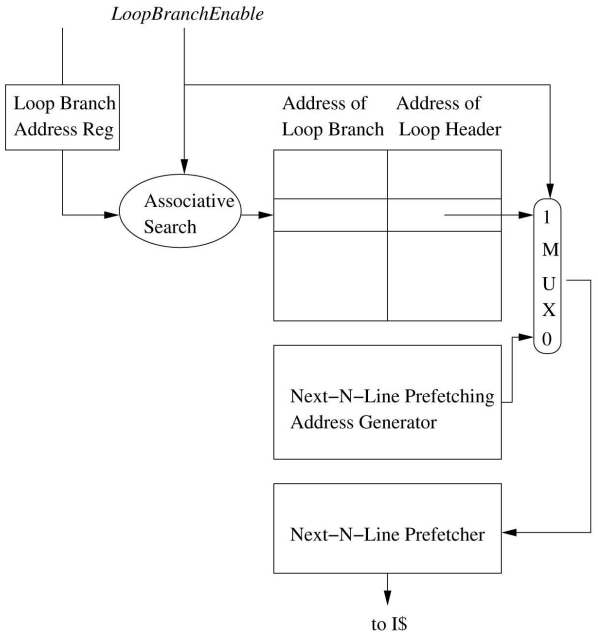


Figure 2.2. The architectural support for the loop-directed instruction prefetching.

instruction is being executed, its address will be passed to the *loop branch address* register, and the *LoopBranchEnable* signal will be enabled to 1 (normally, *LoopBranchEnable* is 0 for non-loop-branch instructions). As we can see from Figure 2.2, the *LoopBranchEnable* signal will enable the associative search circuit to find the corresponding loop header address (i.e., the target address of the loop branch) in the hardware table. This loop header address is then passed to the multiplexer that is controlled by the *LoopBranchEnable* signal. Since the *LoopBranchEnable* signal is enabled, the hardware prefetcher will prefetch instructions from the loop header instead of the next instruction (i.e., PC+4) after the loop branch. When a non-loop-branch instruction is executed, however, the *LoopBranchEnable* signal will be disabled. In that case, the loop-directed

prefetcher will prefetch sequential instructions according to the Next-N-Line prefetching address generator. It should be noted that for a processor that employs branch prediction, the hardware overhead of the loop directed prefetching can be further reduced, because the hardware table shown in Figure 2.2 actually functions like a branch target address (BTB) table (but only for loop branches) and thus can reuse the branch prediction hardware.

2.3 WORST-CASE TIMING ANALYSIS OF LOOP-DIRECTED INSTRUCTION PREFETCHING

The WCET analysis of the loop-directed instruction prefetching is based on the static cache simulation [7, 8] and a recent work in [19]. To better understand our approach, first we give a brief overview of the static cache simulation in Section 2.3.1. Then, we introduce our approach to categorizing instruction accesses and calculating WCET with the loop-directed prefetching in Sections 2.3.2 and 2.3.3, respectively. Finally, an example is discussed in Section 2.3.4.

2.3.1 Background on Static Cache Simulation

To bound the worst-case performance of instruction caches, Arnold et al. [7, 8] proposed static cache simulation to statically categorize the caching behavior of instructions into four different categories based on their conditions. These four categories are summarized below:

1. *Always hit*: A reference to an instruction is always hit if this instruction is guaranteed to be always in the cache when it is accessed.

2. *Always miss*: A reference to an instruction is always miss if this instruction is guaranteed to be not in the cache when it is accessed.

3. *First hit*: A reference to an instruction in a loop is first hit if the first access to this instruction is a hit while all remaining references to this instruction are guaranteed to be misses.

4. *First miss*: A reference to an instruction in a loop is first miss if the first access to this instruction is a miss while all remaining references to this instruction are guaranteed to be hits.

Given a program, the static cache simulation performs control flow analysis and calculates abstract cache states associated with each basic block and loop. Based on the classified instruction categories, timing analysis can be conducted to compute the worst case performance of instruction caches. It is shown in [7, 8] that using an instruction cache can achieve much better performance than a processor that simply disables the instruction cache. In addition, the performance bound that can be estimated is also improved. More detailed information about static cache simulation can be found in [7, 8].

2.3.2 Categorizing Instruction Accesses with Loop-Directed Prefetching

The loop-directed instruction prefetching can have various impacts on the instruction caching behavior. For instance, an *always miss* instruction can be turned into *always hit* if it is guaranteed to be always prefetched into the

instruction cache before it is used. On the other hand, prefetching instructions too early or too late may pollute *always hit* instructions, which may change their status to either *first miss* or *always miss*. To accurately classify the instruction references into the aforementioned four categories with the use of the loop directed prefetching, we design an algorithm by extending the recent work in [19]. As can be seen in Figure 2.3a, this algorithm is composed of three phases, including initialization, loop analysis, and branch analysis. The input of our algorithm is a region [21], which can be a procedure, a loop, or a basic block. The Initialization phase initializes the status and latency of each instruction based on the code placement in the cache line as well as the prefetching distance. More specifically, the first instruction in the cache line is initially classified as *always miss*, while the rest of instructions in the same cache line are *always hit* due to the spatial locality. To take into account the impact of prefetching on the latency of instruction accesses, a variable v_clk is used to record the number of clock cycle saved (i.e., stall cycles reduction) due to the loop-directed instruction prefetching. As can be seen in Figure 2.3b, if the v_clk associated with an instruction is larger than or equal to the instruction cache miss penalty, this instruction will be identified as *always hit* since it can be always prefetched into the cache before it is needed.

The algorithm of loop analysis is described in Figures 2.3c and d, whose task is to update the status and latency of each instruction within the loop by considering the repetition of instruction accesses in loops. The branch analysis is shown in Figure 2.3e, which deals with the status and timing of branch operations

in the program, including the loop branches. Basically, the non-fall-through target of each branch is analyzed, and its status and latency are calculated and updated for a given prefetching distance.

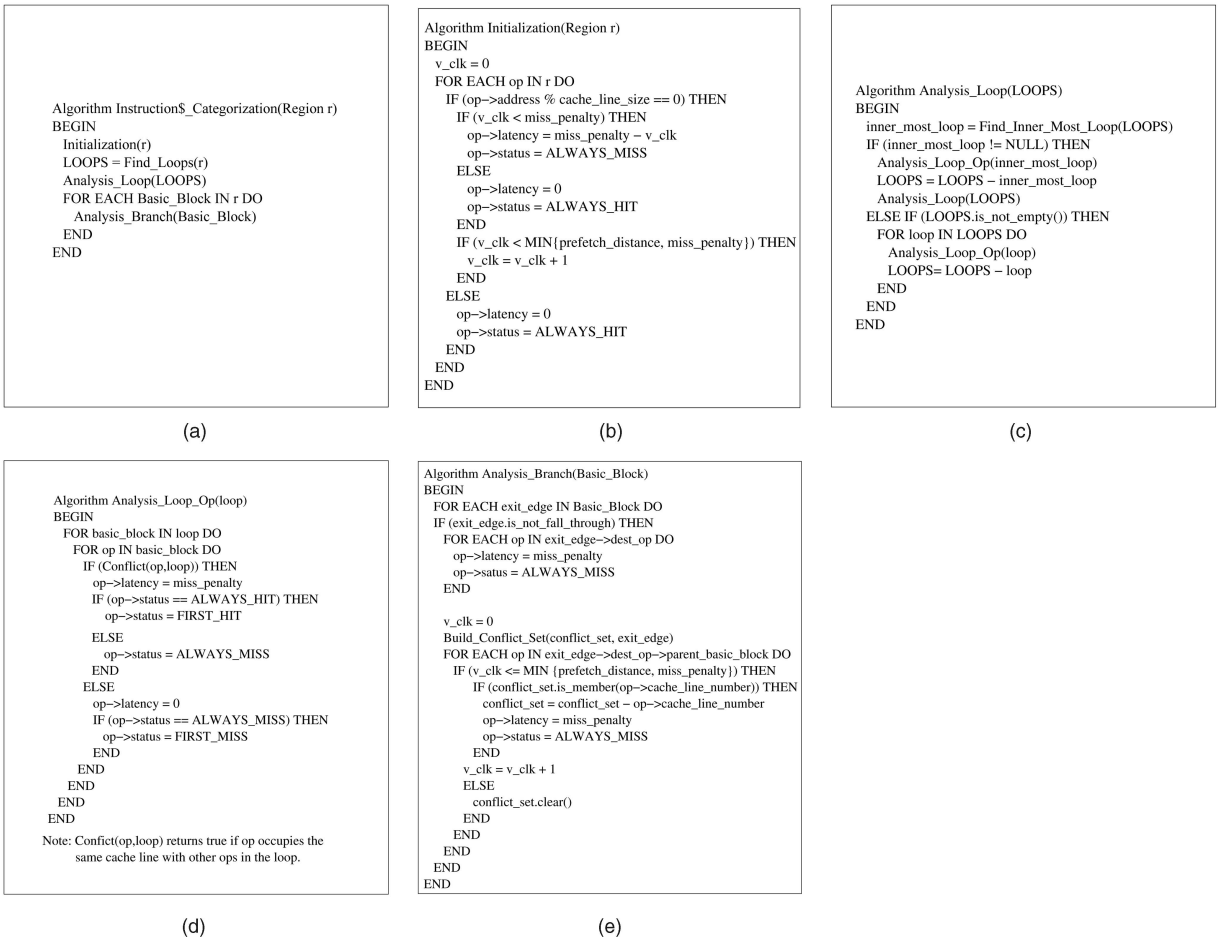


Figure 2.3. Algorithm of categorizing worst-case instruction cache behaviors with the loop-directed instruction prefetching. (a) Main function. (b) Initialization. (c) Loop analysis. (d) Loop op analysis. (e) Branch analysis.

2.3.3 Calculate WCET

Based on the instruction categorization results, the WCET can be calculated similar to the algorithm presented in [19]. More specifically, the worst-case

performance with loop-directed instruction prefetching is computed as the sum of computing cycles and instruction cache stall cycles, because in statically-issued architecture, such as VLIW (which is our target processor), the whole instruction pipeline must be stalled in case of instruction cache misses. The computing cycle is the worst-case execution cycles by assuming a perfect instruction cache, which is the product of scheduled time length and control frequency of each block that can be obtained from the compiler. [3]

The number of instruction cache stall cycles is determined by the cache categorization and the weight of each instruction. Specifically, for an *always miss* instruction, *stalls* are calculated as the product of its I-cache access latency and the weight of that instruction. For a *first hit* instruction, stalls are the product of its latency and *(weight - 1)* of this instruction. For an instruction categorized as *first miss*, the latency of this instruction is added into *stalls* only once. Finally, for *always hit* instructions, their *stalls* are simply 0.

2.3.4 An Example

To illustrate the advantage of the loop-directed prefetching, a code segment is selected from a real-time benchmark called Fib_call [22], whose source code and assembly code (based on the HPL-PD architecture [23]) are shown in Figures 2.4a and 2.4b, respectively. This code segment contains one loop and two basic blocks,

[3] In this work, we assume the maximal number of loop iterations can be analyzed by the compiler or specified manually, which is also supported by SNU real-time benchmarks [22] that will be used in our evaluation.

and its control-flow graph (including each instruction) is shown in Figure 2.4c.

For illustration purpose, we assume an instruction cache with two cache lines, and each line can store two instructions. As can be seen in Figure 2.4d, the assumed instruction cache miss latency is 3 cycles and the prefetching distance is 4. Without any prefetching, the status (i.e., cache categorization) of each instruction is shown in Figure 2.4d. As we can see, there are six instructions inside the loop, among which op9 and op13 are mapped to the same cache line. Therefore, based on our cache categorization algorithm given in Figure 2.3, both op9 and op13 are categorized as *always miss*, and other loop instructions are identified as either *always hit* or *first miss*.

Figure 2.4e shows the effects of the traditional Next-N-Line prefetching (i.e., non-loop-directed). Since the instruction cache miss penalty is 3, all the instructions that can be prefetched before they are used become either *always hit* (e.g., op7) or *first hit* (e.g., op9). Nevertheless, based on the Next-N-Line prefetching, when op14 (i.e., the loop branch) is being executed, four more cache lines will be prefetched, including op15-20. Unfortunately, op15 conflicts with op11; and op17 conflicts with op7. As a result, the status of op11 is changed from *first miss* (without prefetching) to *first hit* (with the Next-N-Line prefetching), which will actually increase the instruction cache misses. For op9, since it is classified as always miss without prefetching, the additional conflict between op9 and the prefetched op17 will not aggravate it.

The instruction categorization with the loop-directed prefetching is

demonstrated in 2.4f. With the loop-directed prefetching, op15 will not be prefetched while the loop branch op14 is encountered. Hence, the status of op11 is converted from *first hit* (with the Next-N-Line prefetching) to *always hit* (with the loop-directed prefetching), leading to better cache performance.

Notice that in this example, op13 changes from Always Miss (without prefetching) to First Hit (with both forms of prefetching). This is because that with both NLP and LP schemes, the instructions are prefetched sequentially before encountering loop branches, and the prefetching distance (4) is larger than the miss penalty (3); therefore, op13 is already prefetched into the cache before it is executed. Thus, op13 can be only classified as either First Hit or possible Always Hit. However, op13 can not be classified as Always Hit with both prefetching schemes. With NLP, only when op9 is executed for the second time (i.e., after the loop branch is taken), the prefetcher will begin to prefetch the next four cache blocks, including op13. However, assuming the processor fetches one cache line each cycle, op13 needs to be fetched two cycles later (op9-10 and op11-12) while it is not in the cache yet because the miss penalty is 3 cycles. Thus, op13 is classified as First Hit. With loop-based prefetching, while op9 can be prefetched while op13 is being executed, op9 is still a miss after the first time it is executed, although the miss latency can be reduced by one cycle. Similar to NLP, op13 is only prefetched when op9 is executed for the second time, which is too late to fill op13 into the cache before it is executed again. Therefore, op13 is still classified as First Hit.

For this code segment, the average-case (i.e., obtained through simulation)

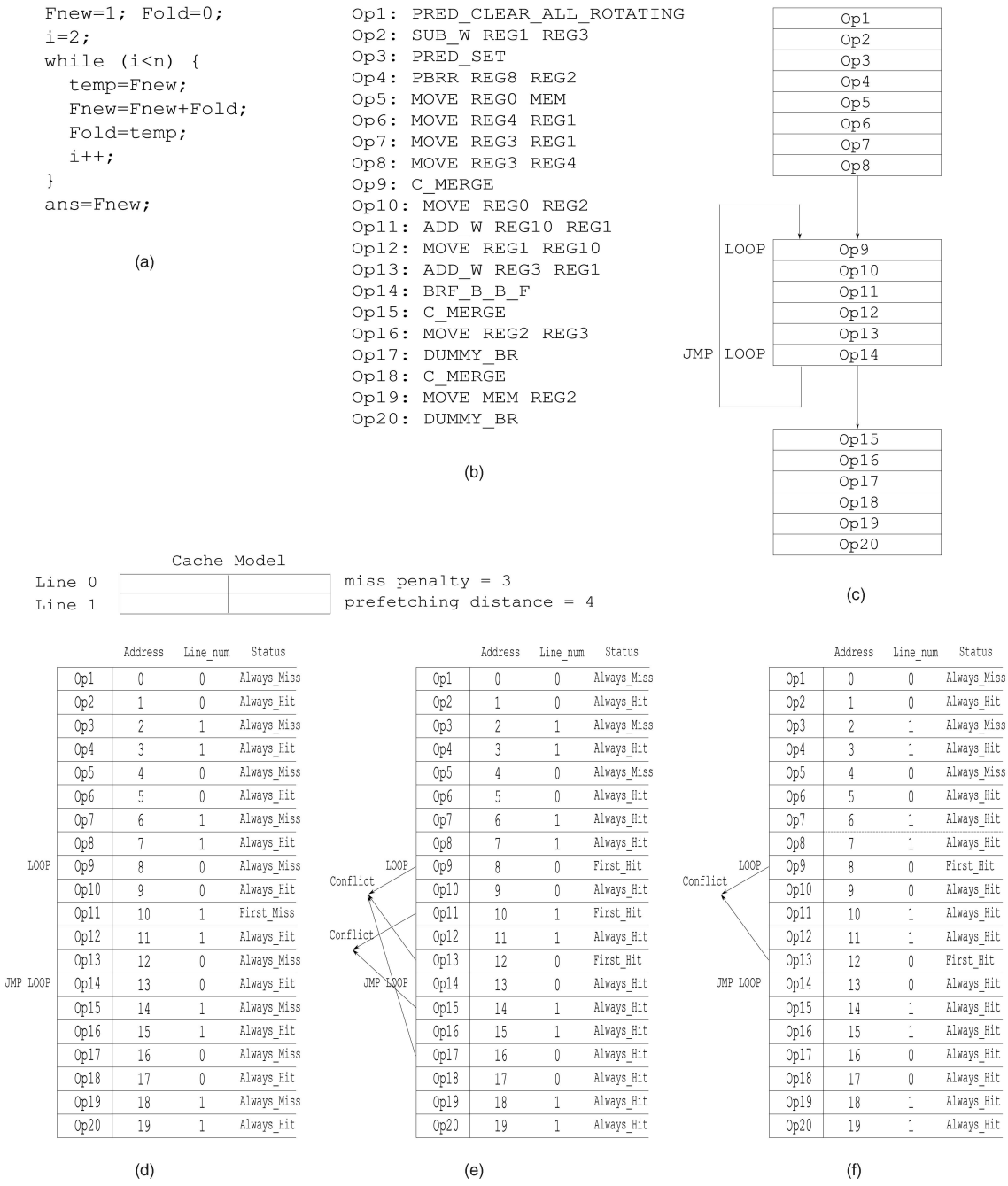


Figure 2.4. An example. (a) Source code. (b) Assembly code. (c) Control_flow graph. (d) Without prefetching. (e) Next-N-Line prefetching. (f) Loop-directed prefetching.

and the worst-case (i.e., obtained through the analytical technique) cache performance of different prefetching schemes are compared in Table 2.2; in which the base scheme represents the method that does not use any prefetching, Non-Loop-directed Prefetching (NLP) refers to the Next-N-Line prefetching [9, 10], and LP stands for the Loop-directed Prefetching. More details of our evaluation method can be found in Section 2.4. Note that the sources of differences between the average-case and worst-case performance typically include conditional branches, overestimated loop bounds and overestimated architectural timing such as cache misses. As we can see, both the average-case and worst-case instruction cache miss rates of the Next-N-Line prefetching are worse than those of the base scheme, due to the adverse effects of cache pollution by the Next-N-Line prefetching. By comparison, the loop-directed prefetching is superior to both the base and the Next-N-Line prefetching in terms of the average case and worst-case instruction cache misses. However, it should be noted that although the Next-N-Line prefetching may increase the instruction cache misses compared with the base scheme, it (as well as the loop-directed prefetching) may reduce the access latencies of missed instructions through prefetching, which can positively impact the overall performance.

2.4 EVALUATION METHODOLOGY

We study the worst-case and average-case performance of the loop directed prefetching and the Next-N-Line prefetching [9, 10] on a VLIW processor based on the HPL-PD architecture [23] by using Trimaran compiler/simulator infrastructure

Table 2.2. The Average-Case and Worst-Case Cache Performance of Different Schemes for the Code Given in Figure 2.4

Schemes	I$ Accesses	I$ Misses	Miss Rate
Base (Average-Case)	194	58	29.89%
Base (Worst-Case)	194	58	28.89%
NLP (Average-Case)	194	81	41.75%
NLP (Worst-Case)	194	81	41.75%
LP (Average-Case)	194	54	27.83%
LP (Worst-Case)	194	54	27.83%

[24]. The average-case performance is obtained through simulation, and the worst-case results are obtained through the analytical technique. We have modified both the back-end compiler Elcor and the simulator to support the loop-directed instruction prefetching. The WCET analysis described in Section 2.3 has been implemented as independent modules to report the worst-case performance. The important parameters of the baseline VLIW processor are given in Table 2.3. Note that to limit the scope of this study, we assume the data cache is perfect, which is also assumed in [19].

For this evaluation, we randomly select six benchmarks from the SNU real-time benchmark suite [22] and two benchmarks (i.e., cordic and rawcaudio) from Mediabench [25]. All the benchmarks are compiled by using the Trimaran compiler. The front-end compiler Impact uses optimization level 4 (O4), and the back-end compiler Elcor uses basic block scheduling and region-based register allocation. The salient characteristics of the benchmarks are shown in Table 2.4. Note that our experiments show that on average, the overestimation of the

Table 2.3. Configuration Parameters and Their Values in the Base Configuration of the Simulated VLIW Processor

Configuration Parameter	Value
Processor	
Function Units	2 integer FUs
	2 floating-point FUs
	1 load/store unit
	1 branch unit
Register File	16 global registers
Cache and Memory Hierarchy	
L1 Instruction Cache	512 bytes, direct-mapped, 8 byte blocks
	1 cycle latency
L1 Data Cache	perfect
Memory	8 cycle, unlimited size

estimated WCET as compared to the observed WCET through simulation is only 9.7 percent. Thus, we believe our WCET analyzer is reasonably tight.

2.5 EXPERIMENTAL RESULTS

2.5.1 Impact On Worst-Case Performance

Figure 2.5 compares the worst-case performance between the Next-N-Line prefetching and the loop-directed prefetching with the prefetching distance varying from 2 to 4, 8, and 16, which is normalized with the WCET of the base scheme that does not use any instruction prefetching. We use NLP-i (or LP-i) to represent the Next-N-Line prefetching (or loop-directed prefetching) with a prefetching distance i. As we can see from Figure 2.5, both the NLP and LP schemes improve the worst-case performance in most cases, except when the prefetching distance is

Table 2.4. The Salient Characteristics of the Selected SNU Real-Time Benchmarks

Benchmark	Description	Static Instrs	I$ accesses	I$ misses	I$ Miss Rate
Bmm	Multiplies two matrices	522	101157	293	0.29%
Fib_mem	Computes a Fibonacci number using a linear recurrence	93	237	41	17.30%
Nested	Sum up the elements in a two-dimensional array	120	2860	76	2.66%
Fibcall	Fibonacci series function	43	208	25	12.02%
Ludcmp	LU decomposition algorithm	265	3799	360	9.48%
Matmul	Matrix multiplication	186	2838	58	2.04%
Cordic	Timing sensitivity stress mark	898	4652240	1934866	41.6%
Rawcaudoio	Speech compression and decompression algorithms	489	10263149	1425463	13.9%

too large (e.g., NLP-16 and LP-16 for Fibcall). In particular, both the NLP and LP schemes are particularly successful for benchmarks that suffer from more instruction caches misses, for instance Cordic and Fib_mem, whose I-cache miss rates are 41.6 and 17.3 percent, respectively, as given in Table 2.4.

Generally, we observe that when the prefetching distance increases from 0 (i.e, base) to 2, 4, and 8, the number of worst case execution cycles is reduced. However, when the prefetching distance increases beyond 8, on average, both the Next-N-Line prefetching and the loop-directed prefetching result in worse WCET, due to the aggravated instruction cache pollution by prefetching too many instructions. By comparing the NLP scheme with the LP scheme with the same prefetching distance, we observe both schemes have very similar worst-case performance for a small prefetching distance such as 2 or 4. This is because when the prefetching distance is smaller than the cache miss penalty (i.e., 8 cycles), prefetching alone cannot translate a cache miss into a hit. However, with larger

prefetching distance (e.g., 8 or 16), we find the LP scheme outperforms the NLP scheme for all the benchmarks. The reason is that the LP scheme can mitigate the cache pollution caused by the prefetched instructions outside loops and prefetch the right instructions for loop execution. Particularly, the best loop directed prefetching scheme (i.e., LP-8) can reduce the base WCET by 23.5 percent on average, which is 3.8 percent more than that of the best Next-N-Line prefetching scheme (i.e., NLP-8).

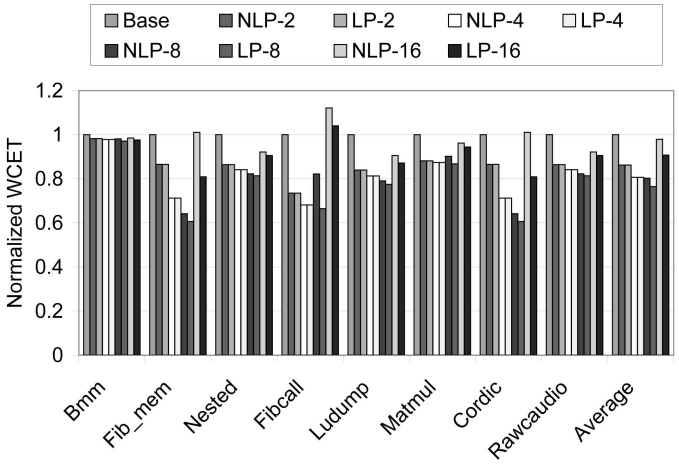


Figure 2.5. Normalized worst-case execution cycles by increasing the prefetching distance from 2 to 4, 8, and 16 for the Next-N-Line prefetching and the loop directed prefetching, which are normalized with the worst-case execution cycles of the Base (without instruction prefetching).

Figure 2.6 compares the worst-case instruction cache miss rates for both the Next-N-Line and loop-directed prefetching with the prefetching distance varying from 2 to 4, 8, and 16, which are normalized with the base I-cache miss rate. As can be seen, when the prefetching distance is less than 8, both the Next-N-Line prefetching and the loop-directed prefetching have the same I-cache miss rate as

the base scheme. This is because when the prefetching distance is smaller than the instruction cache miss penalty (which is 8 cycles), instruction prefetching cannot convert a cache miss into a cache hit, though it may reduce the penalty of that cache miss. For both prefetching approaches, the best instruction cache miss rates are achieved when the prefetching distance is 8, which is the same as the I-cache miss penalty. When the prefetching distance is 16 (i.e., or generally larger than the I-cache miss penalty), too many prefetched instructions may pollute the instruction cache, leading to worse I-cache miss rate. For example, for the NLP scheme, when the prefetching distance is 16, the worst-case I-cache miss rates of Fib_mem and Fibcall are increased by 24 and 55 percent, respectively. This explains why the estimated worst-case execution time of these two benchmarks is worse than the base WCET, as shown in Figure 2.5. Similarly, the LP-16 scheme increases the worst case I-cache miss rate of Fibcall by 43 percent, which is why Fibcall has bad WCET with the LP-16 scheme in Figure 2.5.

2.5.2 Impact on Average-Case Performance

In addition to the worst-case performance, we also compare the loop-directed prefetching and the Next-N-Line prefetching in terms of the average-case performance (i.e., simulated cycles), which are given in Figure 2.7. In general, we observe that for both schemes, the best average-case performance is achieved when the prefetching distance is 2 (note Fibcall and Ludump are the two exceptions, whose best performance results are achieved when the prefetching distance is 4). These average-case performance results are in contrast to the best WCET that can

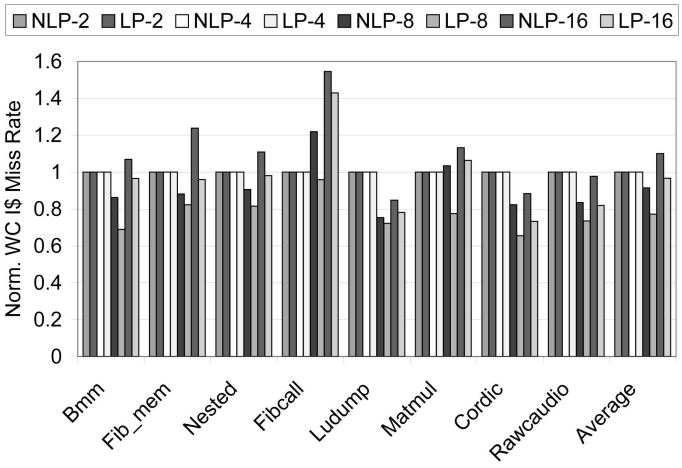


Figure 2.6. The normalized worst-case instruction cache miss rates of the Next-N-Line prefetching and the loop-directed prefetching with the prefetching distance varying from 2 to 4, 8, and 16, which are normalized with respect to the base worst-case instruction cache miss rate.

only be attained with a larger prefetching distance (i.e., 4 for NLP and 8 for LP), as shown in Figure 2.5. The reason is that the cache pollution can be accurately evaluated in a simulator, while it often has to be overestimated by the WCET analyzer due to the lack of precise runtime information. Therefore, when the prefetching distance is larger (but smaller than or equal to the I-cache miss penalty, i.e., 8), the number of I-cache misses (when prefetch distance is 8) and/or the latencies of missed instructions (when prefetching distance is 4 or 8) can be statically estimated as decreased, thus potentially leading to better estimated worst-case performance. By comparison, when the prefetching distance is beyond 2, our simulation indicates that the I-cache miss rate becomes to grow dramatically for all the benchmarks except Fibcall and Ludump, as shown in Figure 2.8. This explains why those benchmarks can achieve the best average-case performance

when the prefetching distance is 2. As can be seen from Figure 2.8, for Fibcall and Ludump, the I-cache miss rate is decreased (dramatically for Ludump) as the prefetching distance increases from 2 to 4 with the Next-N-Line prefetching. This is why for these two benchmarks, the best average-case performance with the Next-N-Line prefetching is achieved when the prefetching distance is 4.

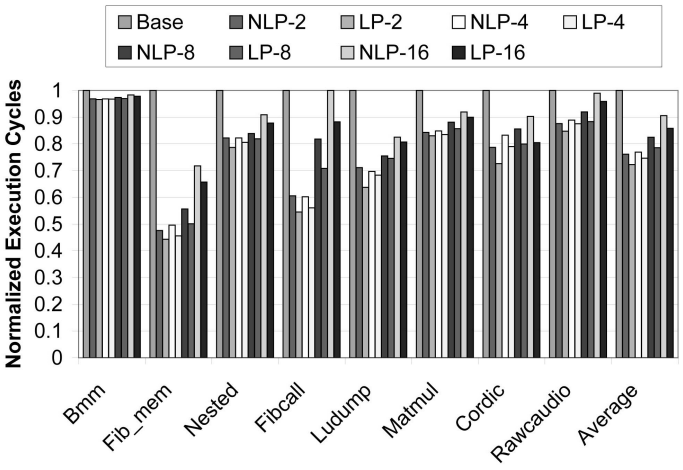


Figure 2.7. Normalized execution cycles by increasing the prefetching distance from 2 to 4, 8, and 16 for both the NLP and LP schemes, which are normalized with the base execution cycles without instruction prefetching.

2.5.3 Sensitivity to the Cache Size

We have also made experiments to study the effects of both the Next-N-Line prefetching and the loop-directed prefetching on instruction caches with different sizes. Figs. 9 and 10 show the averaged execution cycles and the averaged WCET respectively for both prefetching schemes with the I-cache size reduced from 512 to 256 and 128 bytes, which are normalized with the execution cycles and the WCET respectively of the base scheme with a 512-bytes instruction cache. As one can

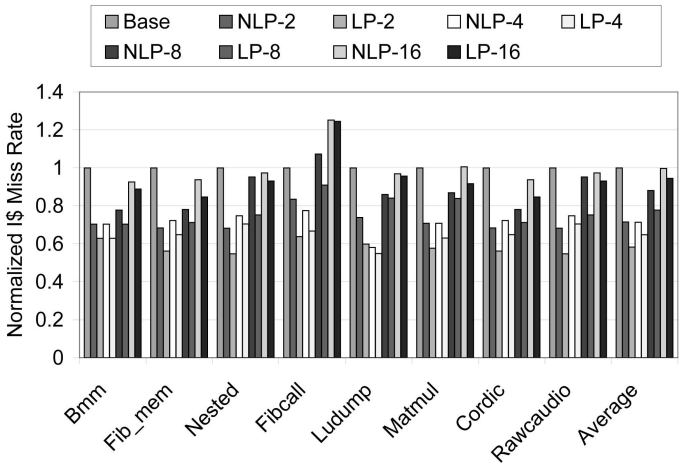


Figure 2.8. Simulated instruction cache miss rates of NLP and LP schemes by increasing the prefetching distance from 2 to 4, 8, and 16, which are normalized with the base execution cycles without instruction prefetching.

expect, when the I-cache size is reduced, especially from 256 to 128 bytes, the average-case as well as the worst-case performance generally decreases. For a smaller instruction cache such as a 128-byte I-cache, a long prefetching distance (e.g., 16) can significantly degrade both the average-case and worst-case execution time, and only a small prefetching distance (i.e., 2) can benefit performance. This is because cache pollution by prefetching many instructions becomes more severe in smaller instruction caches.

We also find that while both the Next-N-Line prefetching and the loop-directed prefetching with a proper prefetching distance are useful to enhance performance, on average, both techniques are more effective for larger instruction caches. For instance, NLP-2 and LP-2 increases the average-case performance of the 512-byte instruction cache by 23.9 and 27.7 percent, respectively, while the improvement on the 128-byte instruction cache is only 13.2 and 17.2 percent,

respectively. Similarly, as we can see in Figure 2.10, the maximal WCET reduction for a 512-byte I-cache is 19.7 (by NLP-8) and 23.5 percent (by LP-8), whereas the best WCET reduction for a 128-byte I-cache is only 3.7 percent (by NLP-2) and 6. The reason is that for a very small cache, even with a moderate prefetching distance, the prefetched instructions are more likely to replace other useful instructions. In contrast, a larger cache can accommodate more prefetched instructions to benefit performance.

Interestingly, as can be seen in Figure 2.9, a 256-bytes I-cache can exploit either the Next-N-Line prefetching (e.g., NLP-2, NLP-4, or NLP-8) or the loop-directed prefetching (e.g., LP-2, LP-4, LP-8, or LP-16) to achieve performance better than a 512-bytes I-cache (i.e., the base), which demonstrates the effectiveness of these instruction prefetching techniques and the importance of tuning the prefetching distance to achieve the best performance improvement. In addition, as we can observe from both Figures 2.9 and 2.10, the loop directed prefetching always outperforms the Next-N-Line prefetching in terms of both the average-case and the worst-case performance, indicating that the loop-directed prefetching is a better instruction prefetching technique for real-time applications.

2.6 CONCLUSION

In this chapter, we propose a loop-based instruction prefetching scheme to enhance the performance for real-time applications. Compared with the Next-N-Line prefetching [9, 10], the loop directed approach can mitigate cache pollution by not prefetching instructions after the loop branches and can enhance

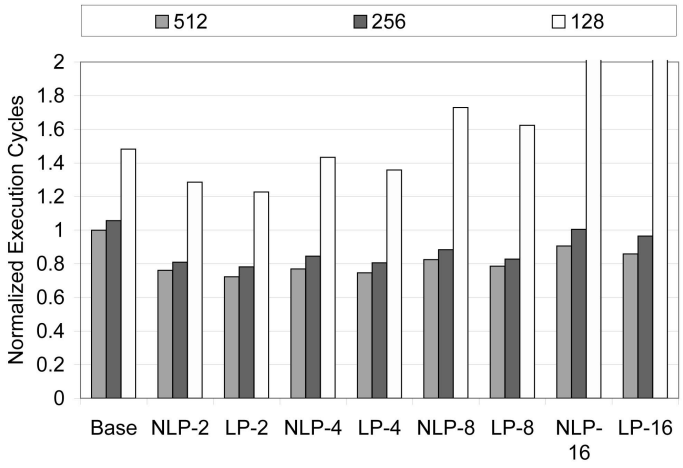


Figure 2.9. Averaged execution cycles of the NLP and LP schemes with different prefetching distances when the instruction cache size is reduced from 512 to 256 and 128 bytes, which are normalized with respect to the execution cycles of the base scheme with a 512-bytes instruction cache.

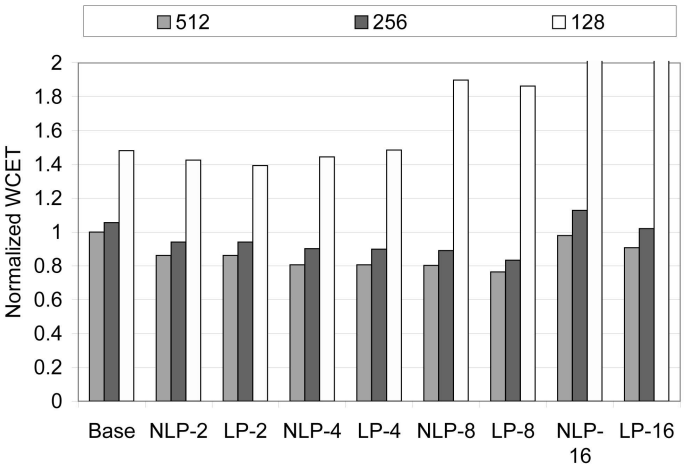


Figure 2.10. Averaged WCET of the NLP and LP schemes with different prefetching distances when the instruction cache size is reduced from 512 to 256 and 128 bytes, which are normalized with respect to the WCET of the base scheme with a 512-bytes instruction cache.

performance by prefetching the right instructions during the loop execution. The architectural and compiler support for the loop-directed prefetching is simple and cost-efficient. Built upon prior work in WCET analysis [7, 8, 19], we present an approach to modeling the loop directed prefetching and estimating the worst-case performance for instruction caches with the loop-directed prefetching.

Our experimental and static analysis results indicate that the loop-directed prefetching can achieve both better average-case and worst-case performance than the Next-N-Line prefetching, and thus is preferable for real-time applications. We also observe that the prefetching distance has large impact on the average-case as well as the worst-case performance; however, a prefetching distance resulting in the best average-case performance does not automatically lead to the best WCET. Actually, our evaluation shows that the best prefetching distance for the worst-case performance is slightly longer (but not too long as compared to the instruction cache miss penalty) than the best prefetching distance for the average-case performance. The reason is that the cache pollution caused by the prefetched instructions can be accurately evaluated in a simulator, while it often has to be overestimated by the WCET analyzer due to the lack of precise runtime information. On the other hand, the WCET analyzer can statically estimate the benefits of prefetching with a longer distance, for instance, the reduced number of I-cache misses (in case the prefetching distance is longer than or equal to the I-cache miss latency) as well as the decreased penalty for missed instructions. Consequently, for real-time applications, the best prefetching distance must be

selected based on the worst-case timing analysis, not simply based on the average-case results through simulation.

CHAPTER 3

MULTICORE-AWARE CODE POSITIONING TO IMPROVE WORST-CASE PERFORMANCE

3.1 CHAPTER OVERVIEW

With the rapid development of computing technology and the diminishing return of completed uniprocessors, multi-core chips processors have been increasingly adopted. Presently, multi-core processors have been widely utilized in all types of computer systems, such as high performance general-purpose servers, specialized embedded systems and so on. In particular, with the growing demand of high performance by high-end real-time applications such as HDTV and real-time multimedia processing applications, multi-core processors are expected to be increasingly used in the real-time systems. Actually, researchers have envisioned that the real-time systems will be possibly deployed on large-scale multi-core processors which are composed of tens or even hundreds of cores on a single chip in the near future [50].

For real-time systems, it is critical to accurately obtain the worst-case execution time for each task, which provides the basis of task scheduling. Besides, optimizing real-time code to reduce WCET can bring many benefits to real-time systems. For instance, better WCET of a task gives the real-time scheduler more flexibility to schedule this task for meeting its deadline. Also, reducing WCET of a computing task can help conserve power used by the system [44]. The basic idea is

that with WCET information available, if a task still have slacks, the clock rate can be lowered to reduce power dissipation while still meeting the deadlines.

To reduce the WCET of real-time tasks (i.e., to obtain "better" WCET), code positioning approaches have been proposed [31, 32]. However, current WCET-oriented code positioning approaches center on enhancing the WCET of single-threaded application on the uniprocessors, which cannot be effectively applied to multi-core processors with shared caches. This is because these code positioning algorithms [31, 32] only reduce the intra-thread cache conflicts, but can not detect the inter-thread cache conflicts or avoid them. Furthermore, these approaches may reduce the intra-thread L1 cache misses at the cost of more inter-thread shared L2 cache misses, whose penalty is usually much more than that of an L1 cache miss and thus may hurt the overall performance. Therefore, it is crucial to develop multicore-aware code positioning techniques for real-time applications running on multicore platforms.

In this chapter, we assume two real-time threads are running concurrently on a dual-core processor with a shared L2 cache and our goal is to reduce the WCET of these threads [1]. We have studied three approaches — a worst-case-oriented approach (WCO) and two fairness-oriented approaches, including the

[1]In some applications with mixed real-time and non-real-time tasks, a real-time thread may run concurrently with a non-real-time thread. However, it should be noted that code position for this scenario is actually less challenging, as the performance of the non-real-time thread can be sacrificed for enhancing the WCET of the real-time thread [52]. While in this chapter, the WCETs of both real-time threads need to be considered.

percentage-fairness-oriented (PFO) and amount-fairness-oriented (AFO) schemes,all of which are built upon multicore cache WCET analysis, but with different optimization goals. Our experiments show that all these three proposed techniques can effectively reduce the WCET for co-running real-time threads to achieve their respective optimization goals.

The rest of this chapter is organized as follows. Section 3.2 reviews related work. Section 3.3 describes the proposed multicore-aware code positioning approaches. The evaluation methodology is explained in Section 3.4 and the experimental results are presented in Section 3.5. Finally, the conclusions are made in Section 3.6.

3.2 RELATED WORK

Traditional code positioning algorithms mostly aim at enhancing the average-case execution time (ACET) by reordering the basic blocks to make the most frequently traversed edges contiguous in memory [38, 39, 40, 41, 42]. However, as the most frequently traversed edges may not be a part of the worst-case paths, the WCET can not be guaranteed to be reduced by these approaches. Even if the WCET path is taken into account by the code positioning algorithm, a change in the positioning may result in a different path becoming the WCET path.

To improve the worst-case performance in a processor with instruction caches, a code positioning approach is proposed to focus on positioning the basic blocks on the worst-case path in the program to reduce the pipeline delay caused

by the transfer of controls [31]. The main idea of this basic block positioning algorithm is to select edges between basic blocks on the worst-case path to be contiguous, which will minimize the WCET. Recently, another WCET-oriented approach is proposed to reduce the number of cache conflict misses by means of placing procedures which contributes to the WCET, so that they are mapped contiguously in memory layout and the placement avoids overlapping of cache lines belonging to a caller and a callee procedure [32]. Both these two approaches, however, have not considered the inter-thread cache conflicts in multi-core computing platforms.

Cache partitioning is another useful method to isolate tasks in a multitasking real-time system. It allows individual analysis of cache behavior and thus enhances the time predictability of each task. There are mainly two types of cache partitioning approaches, i.e. hardware-based [53, 54] and software-based [55, 56]. In hardware-based cache partitioning, address mapping hardware is inserted into the processor with a cache to restrict cache accesses to a single contiguous cache segment at any one time; therefore, each task has the right to access a private cache segment for one or more partitions. In contrast, the software-based approach creates a private cache partitioning for each task by assigning it a separate address space in the cache with the use of the compiler and the linker. Our multicore-aware code positioning techniques are complementary to cache partitioning approaches. Multicore-aware code positioning enables different tasks to still share caches for achieving benefits such as efficient cache space usage,

low-cost cache coherency and easy sharing [57], while minimizing the inter-core cache conflicts. Moreover, as a pure software-based technique, multicore-aware code positioning does not need to modify the hardware while achieving "better" WCET for real-time tasks running on multi-core processors.

3.3 OUR APPROACHES

3.3.1 Overview

In a multi-core processor, each core typically has private L1 instruction and data caches. The L2(and/ or L3) caches can be either shared or separated. While private L2 caches are more time-predictable in the sense that there are no inter-core L2 cache conflicts, they suffer from other deficiencies. First, each core with a private L2 cache can only exploit separated and limited cache space. Due to the great impact of the L2 cache hit rate on the performance of multi-core processors, private L2 caches may have worse performance than a shared L2 cache with the same total size, because each core with shared L2 cache may make use of the aggregate L2 cache space more effectively. Besides, separated L2 caches increase the cache synchronization and coherency cost [57]. Moreover, a shared L2 cache architecture makes it easier for multiple cooperative threads to share instructions and data, which becomes more expensive in separated L2 caches [57]. Therefore, we will study the WCET analysis of multi-core processors with shared L2 caches in this chapter.

For simplicity, we assume that two real-time threads run concurrently on different cores of a dual-core processor with private L1 caches and a shared L2

cache, although our techniques can be applied or adapted for multiple threads running on multi-core chips with multi-level memory hierarchies. We have proposed three strategies to optimize the WCET of both threads for making different tradeoffs. These strategies include a Worst-Case-Oriented strategy, and two Fairness-Oriented strategies, including both AFO and PFO. The WCO aims at improving the performance of the real-time thread with the longest WCET, as this type of thread mostly impacts the performance of the whole system. AFO and PFO attempt to treat all the real-time threads fairly, that is to optimize the WCET of each real-time thread by approximately an equal amount or percentage respectively.

Figure 3.1 depicts the main working flow of the WCET-oriented co-optimization architecture, which mainly consists of two sub-flows. The sub-flows of both threads are initialized with code analysis including control flow analysis and static cache analysis. The inter-thread cache conflict analysis algorithm calculates the worst-case inter-thread L2 cache conflict set. Then the codes of both threads are positioned following a specific strategy to reduce the inter-thread L2 cache conflicts. The WCET analysis is conducted to calculate the new WCETs for both threads after positioning, which are compared with their original WCETs for guiding the co-optimization further. It is worthy to note that the sub-flow of both threads from code analysis to code positioning may be repeated for several times to achieve the optimal results.

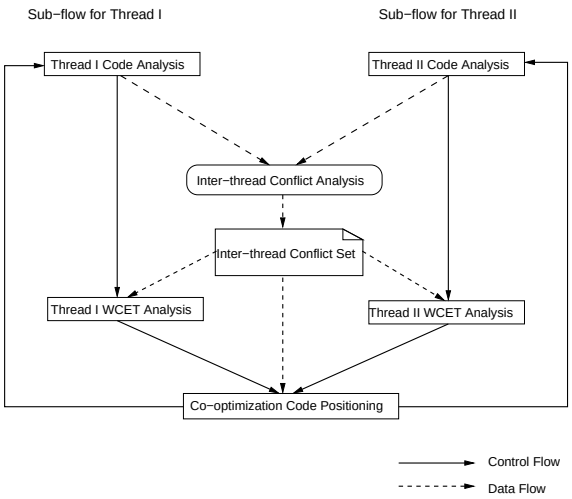


Figure 3.1. Flow diagram of WCET-oriented co-optimization architecture.

3.3.2 Worst-Case-Oriented Code Positioning

The objective of WCO is to minimize the longest WCET of both real-time threads (i.e. reducing the worst-case WCET of co-running threads), whose algorithm is described in Algorithm 1. The inputs of the algorithm are the two programs to be optimized. In line 2, the termination variable of the algorithm is initialized. In the next three lines, fundamental data needed by the algorithm are calculated, including the original WCETs of both programs and the L2 cache conflict instruction list. After the original WCETs of both programs are compared, the program with smaller original WCET will be positioned to optimize the WCET of the other program as much as possible. As shown from line 7 to line 15, in case that the original WCET of P1 is larger than that of P2, P2 will be positioned at line 8, in which the conflict instructions from P2 that lead to the largest inter-thread cache conflicts will be allocated at new memory addresses mapping to L2 cache blocks with the minimal conflicts with the corresponding

instructions from P1. The WCETs of both programs will be calculated again when the positioning of P2 finishes from lines 9 to line 10. If the WCET of P1 is still larger than that of P2, the termination variable will be assigned as true; otherwise, the function of *WC_Oriented_Code_Positioning* will be executed recursively to reduce the WCET of P2, which now becomes the thread with the longest WCET. In the other case (line 16 to 25), the positioned program turns to be P1 as the original WCET of P2 is larger than that of P1, and other steps are almost the same as the first case. Finally, the algorithm will not be terminated until the value of termination variable equals true.

3.3.3 Fairness-Oriented Code Positioning

While WCO focuses on optimizing a single thread that has the worst WCET among co-running threads, FO code positioning aims at optimizing all the co-running threads to ensure fairness. Since the WCETs of both threads may vary significantly, the "fairness" has different meanings and implications, depending on the optimizing objectives. In this work, the FO strategies are divided into two different schemes according to the "fairness" goals, including 1) reducing approximately the same amount of WCET, and 2) reducing approximately the same percentage of WCET. Accordingly, two schemes are named Amount-Fairness-Oriented (AFO) code positioning and Percentage-Fairness-Oriented (PFO) code positioning respectively.

Algorithm 1 *WC_Oriented_Code_Positioning*

```
1: begin
2: boolean terminate = false;
3: P1_wcet = WCETAnalysis(P1);
4: P2_wcet = WCETAnalysis(P2);
5: Conflict_Op_List = Bulid_Conflict_Op_List(P1, P2);
6: repeat
7:     if P1_wcet > P2_wcet then
8:         Positioning(P2, Conflict_Op_List);
9:         P1_wcet = WCETAnalysis(P1);
10:        P2_wcet = WCETAnalysis(P2);
11:        if P1_wcet > P2_wcet then
12:            terminate = true;
13:        else
14:            WC_Oriented_Code_Positioning(P1, P2);
15:        end if
16:    else
17:        Positioning(P1, Conflict_Op_List);
18:        P1_wcet = WCETAnalysis(P1);
19:        P2_wcet = WCETAnalysis(P2);
20:        if P1_wcet < P2_wcet then
21:            terminate = true;
22:        else
23:            WC_Oriented_Code_Positioning(P1, P2);
24:        end if
25:    end if
26: until terminate == true;
27: end
```

Amount-Fairness-Oriented Scheme

Amount-Fairness-Oriented (AFO) code positioning algorithm aims at reducing the WCETs of both co-running threads by approximately equal amount. When the WCO code positioning approach is applied, only the instructions of the thread with shorter (i.e. "better") WCET are positioned to reduce the WCET of the other thread as much as possible. In this case, the amount of WCET reduced by avoiding the inter-thread L2 cache conflicts is the same to both threads; however, the difference of the amount of WCET reduction can be caused by different intra-thread L1 and L2 cache misses due to the WCO code positioning. Therefore, AFO can leverage WCO to decrease the inter-thread cache misses, while it tries to recover some of the positioned instructions in WCO by a procedure named $De-positioning$ to ensure that the intra-thread cache miss penalties of both threads are reduced by approximately the same amount.

The algorithm of AFO is demonstrated in Algorithm 2. The inputs and the initialization phase are the same as WCO. In line 6, the WCO algorithm is invoked to reduce the inter-thread L2 cache misses. In this algorithm, P2 is assumed to be the thread with a larger WCET; therefore, only the instructions from P1 are positioned by WCO. Furthermore, some positioned instructions of P1 are recovered to their original positions by the procedure $De-positioning$ at line 8, and the corresponding instructions from P2 are positioned instead to avoid the inter-thread L2 cache conflicts at line 9. After positioning,the resulting WCETs of both programs are computed at line 10 and 11. Then the difference of WCET

reduction of both programs (i.e. ΔW) is calculated at line 12. If this difference is larger than the difference of last iteration (i.e. $\Delta Original\_W$), then the termination variable is assigned as true; otherwise, the smaller difference is assigned to $\Delta Original\_W$ to further minimize the difference in terms of the amount of WCET reduction for both threads. This algorithm is repeated till the value of termination variable becomes true.

Algorithm 2 *AF_Oriented_Code_Positioning*

1: **begin**
2: **boolean** $terminate = false$;
3: $Original\_P1\_wcet = WCETAnalysis(P1)$;
4: $Original\_P2\_wcet = WCETAnalysis(P2)$;
5: $Conflict\_Op\_List = Bulid\_Conflict\_Op\_List(P1, P2)$;
6: $WC\_Oriented\_Code\_Positioning(P1, P2)$;
7: **repeat**
8: $De - positioning(P1, Conflict\_Op\_List)$;
9: $Positioning(P2, Conflict\_Op\_List)$;
10: $P1\_wcet = WCETAnalysis(P1)$;
11: $P2\_wcet = WCETAnalysis(P2)$;
12: $\Delta W = Calculate\_Amount\_Variation()$;
13: **if** $\Delta W >= \Delta Original\_W$ **then**
14: $terminate = true$;
15: **else**
16: $\Delta Original\_W = \Delta W$;
17: **end if**
18: **until** $terminate == true$;
19: **end**

Percentage-Fairness-Oriented Scheme

While AFO targets approximately the same amount of WCET reduction, PFO aims at about the same percentage of WCET reduction. The principle of Percentage-Fairness-Oriented code positioning is described as the following. In the multi-core processor with a shared L2 cache, the WCET of a thread can be broken into the computation time by assuming perfect caches, the L1 cache miss penalty and the L2 cache miss penalty. The L2 cache miss penalty consists of two parts: the intra-thread L2 cache miss penalty and the inter-thread L2 cache miss penalty. The WCET of a thread can be calculated by Equation 1, where E stands for the computation time without considering cache misses, L_1 is L1 cache miss penalty, and In_L_2 and Out_L_2 represent the intra-thread and inter-thread L2 cache miss penalty respectively.

$$WCET = E + L_1 + (In\_L_2 + Out\_L_2) \tag{3.1}$$

After code positioning, the inter-thread cache conflicts will be decreased; however, the intra-thread cache conflicts both on L1 and L2 caches may increase. Since the computation time E is the same before or after code positioning, the improvement of the WCET after code positioning can be illustrated as Equation 2.

$$\Delta WCET = \Delta Out\_L_2 + \Delta L_1 + \Delta In\_L_2 \tag{3.2}$$

As the goal of PFO is to reduce the WCET of each real-time thread by approximately equal percentage, assuming that there are two threads, i.e., Thread

A and Thread B, Equation 3 can be used to characterize this scheme. In this equation, $WCET_A$ and $WCET_B$ are the original WCETs of Thread A and Thread B, respectively, and $\Delta WCET_A$ and $\Delta WCET_B$ are derived from Equation 2 denoting the change of the WCET for each thread.

$$\frac{\Delta WCET_A}{WCET_A} \approx \frac{\Delta WCET_B}{WCET_B} \tag{3.3}$$

Because the execution time E may vary substantially for different real-time threads, it becomes very hard, if not impossible, to guarantee the same percentage of WCET reduction if E is considered. Also, since the execution time E is insensitive to cache-based optimizations, the PFO scheme focuses on reducing the same percentage of L1 and L2 cache miss penalties for both threads through cooperative code positioning. We also find that while the reduction of inter-thread cache conflict is mutual, the L1 cache misses and L2 intra-thread misses of a thread are heavily dependent on how many instructions are positioned to that thread. Specifically, the more instructions are positioned for a thread, the more possible intra-thread L1 and L2 cache conflicts may occur in that thread. Therefore, in order to reduce the WCETs of both threads by approximately equal percentage, the number of instructions to be positioned for each thread should be inversely proportional to its original WCET as depicted in Equation 4.

$$\frac{Instr\_Num_B}{WCET_A} \approx \frac{Instr\_Num_A}{WCET_B} \tag{3.4}$$

Algorithm 3 illustrates the algorithm of Percentage-Fairness-Oriented code

positioning approach. The inputs of the algorithm are the two programs to be optimized. In line 2, the termination variable is initialized. In the next three lines, the original WCETs of both programs are calculated, and the L2 cache conflict instruction list is determined as well. First, the instructions needed to be positioned for both programs are identified according to the designing principle of PFO at line 7 and line 8. Then both programs are positioned at line 9 and line 10. From line 11 to line 12, the WCETs of both programs are calculated after positioning. Based on the original WCETs and new WCETs of both programs, the WCET percentage variance between these two programs is calculated to determine whether or not the WCET percentage variance after positioning is smaller than the original WCET percentage variance at line 13 and 14. If true, the original WCET percentage variance $\Delta Original\_P$ is assigned to be the most recently calculated WCET percentage variance ΔP at line 17; otherwise, the termination variable is assigned to be true at line 15. This algorithm is repeated till the value of termination variable becomes true.

3.3.4 Inter-thread L2 Cache Conflict Analysis

In the co-optimization architecture depicted in Figure 3.1, inter-thread L2 cache conflict analysis is an important step to identify the worst-case inter-core L2 cache conflicts and the associated instructions from different cores. We propose to leverage Yan et al's recent work in [37] to analyze the worst-case inter-thread L2 cache conflicts. The main steps of this algorithm are described in Algorithm 4.

The inputs of this algorithm are the programs of both the co-running

Algorithm 3 *PF_Oriented_Code_Positioning*

```
1: begin
2: boolean terminate = false;
3: Original_P1_wcet = WCETAnalysis(P1);
4: Original_P2_wcet = WCETAnalysis(P2);
5: Conflict_Op_List = Bulid_Conflict_Op_List(P1, P2);
6: repeat
7:     Pos_Op_List_P1 = Build_Pos_Op_List(ConflictOpList);
8:     Pos_Op_List_P2 = Build_Pos_Op_List(ConflictOpList);
9:     Positioning(P1, Pos_Op_List_P1);
10:    Positioning(P2, Pos_Op_List_P2);
11:    P1_wcet = WCETAnalysis(P1);
12:    P2_wcet = WCETAnalysis(P2);
13:    ΔP = Calculate_Percentage_Variation();
14:    if ΔP >= ΔOriginal_P then
15:        terminate = true;
16:    else
17:        ΔOriginal_P = ΔP;
18:    end if
19: until terminate == true;
20: end
```

threads. Initially, the L2 cache status sets for each thread (i.e., without considering inter-thread conflicts) are calculated for both threads respectively, which identify groups of instructions within the same thread sharing same cache lines. In order to find the worst-case inter-thread instruction interferences from two different threads, we distinguish instructions in loops from those not in loops. Each instruction from each thread is examined, whose L1 cache access behavior can be easily obtained by using static analysis techniques for instruction caches [37] (line 5-6).

If there exists an L1 miss, it is checked where this miss happens (line 7), i.e., in or out of loops. If this miss occurs in a loop, it is necessary to determine whether or not the cache line used by this instruction would be occupied by the instructions from the other thread, and whether or not those instructions are also in a loop. The cache line used by this instruction from Thread I can be found by function $Find\_Cache\_Line$ at line 7, and function $Find\_Conflict\_Op$ at line 8 helps to check if there is any instruction from Thread II using the same L2 cache line. If there is an instruction from Thread II that also uses the same L2 cache line, this instruction will be named as $conflict\_op$, and then be checked in a loop or not at line 10. If the conflicting instruction happens to be in a loop as well, then the worst-case number of conflicts of these conflicting instructions is equal to the smaller one of the worst-case number of access times from these two threads (line 11), which can be obtained from control flow analysis.

The inter-thread L2 cache conflict set is constructed in the format of a matrix, where a row index represents the number of instructions from Thread I,

and a column index denotes the number of instructions from Thread II. The element of this matrix is a cache conflict reference object, which contains the L2 cache line number and the frequency of conflicts. After obtaining the worst-case conflict frequency, a cache conflict reference object is generated and added to the matrix at the place determined by the index number of the conflicting instructions (line 12). If the conflicting instruction from Thread II is not in a loop, the inter-thread L2 cache conflict can happen only once in the worst case. Therefore the frequency attribute of the cache conflict reference object is 1, which is added into the inter-thread L2 cache conflict set by function $Add\_Conflict\_Matrix$ at line 14. Also, if the instruction from Thread I is outside a loop, then the worst-case conflict frequency is only 1 as well (line 18-22). More details about this inter-thread L2 cache instruction interference analysis can be found at [37].

3.3.5 WCET Calculation

The WCET of a real-time task is computed by using the implicit path enumeration technique (IPET) proposed by Li and Malik [58, 59]. In IPET, the WCET of each task is calculated by maximizing the objective function in Equation 3.5, in which c_i is the execution cost of the basic block i, including cache miss penalty, and b_i represents the number of time the basic block i is executed. To legally maximize the objective function, program structural constraints should be taken into account, which are derived from the program's control flow information for each basic block i, as described in Equation 3.6. In this equation, in_edge_i is the sum of the edges entering the basic block i, and out_edge_i is the sum of the

Algorithm 4 *Inter_Conflict_Analysis*

```
1: begin
2: T1_Cache_Pos = Initialize_Cache_Pos(T1);
3: T2_Cache_Pos = Initialize_Cache_Pos(T2);
4: for op in T1 do
5:     if Is_L1_Miss(op) then
6:         if Is_In_Loop(op) then
7:             cache_line = Find_Cache_Line(op, T1_Cache_Pos);
8:             conflict_op = Find_Conflict_Op(T2_Cache_Pos);
9:             if conflict_op! = null then
10:                if Is_In_Loop(conflict_op) then
11:                    weight = Min_Weight(op, conflict_op);
12:                    Add_Conflict_Matrix(op, conflict_op, weight);
13:                else
14:                    Add_Conflict_Matrix(op, conflict_op, 1);
15:                end if
16:            end if
17:        else
18:            cache_line = Find_Cache_Line(op, T1_Cache_Pos);
19:            conflict_op = Find_Conflict_Op(T2_Cache_Pos);
20:            if conflict_op! = null then
21:                Add_Conflict_Matrix(op, conflict_op, 1);
22:            end if
23:        end if
24:    end if
25: end for
26: end
```

edges exiting the basic block, which should be equal to each other.

$$Total\ execution\ time = \sum_{i=1}^{n} c_i \times b_i \tag{3.5}$$

$$\sum in\_edge_i = \sum out\_edge_i = b_i \tag{3.6}$$

When an instruction runs in a multi-core processor with a hierarchical cache memory, its execution time depends on whether the instruction access results in a cache hit or a cache miss. Therefore, the total execution time of a program is heavily influenced by the number of cache misses and the penalty of cache misses. The state of L1 instruction cache accesses for each thread running on a multi-core processor with a shared L2 cache can be derived by static cache analysis. In addition, the state of L2 instruction cache accesses for each basic block, including the potential inter-thread L2 cache conflicts, can be computed by the inter-thread L2 cache conflict analysis algorithm depicted in Section 3.3.4. Therefore, the total number of cache misses can be calculated in Equation 3.7. where b_i denotes the number of times basic block i is executed; $m1_i$ is the number of L1 cache misses of the basic block i; and $m2_i$ and $m2_i^{'}$ account for the number of intra-thread L2 cache misses and inter-thread L2 cache misses of basic block i, respectively.

$$Cache\ misses = \sum_{i=1}^{n} m1_i \times b_i + (m2_i + m2_i^{'}) \times b_i \tag{3.7}$$

Equation 3.8 integrates the penalty of cache misses into the objective function to accurately compute the WCET of the whole program. In this equation,

e_i represents the basic execution latency of basic block i by assuming a perfect cache; $l1_{penalty}$ stands for the L1 cache miss penalty; and $l2_{penalty}$ denotes the L2 cache miss penalty.

$$c_i = e_i + m1_i \times l1_{penalty} + (m2_i + m2_i^{'}) \times l2_{penalty} \tag{3.8}$$

As a result, the WCET of the real-time thread can be calculated by using an ILP (Integer Linear Programming) solver to maximize the objective function 3.5.

3.4 EVALUATION METHODOLOGY

We evaluate the proposed multicore-aware code positioning schemes on a heterogeneous dual-core processor with a shared L2 cache. To achieve better performance, energy efficiency and low cost, embedded applications have increasingly used heterogeneous systems including multiple programmable processor cores, specialized memories, and other components on a single chip [60]. For instance, most hand-held devices now adopt a heterogeneous dual-core architecture that is composed of a DSP (Digital Signal Processing) core and an ARM core. In this chapter, we focus on evaluating the multicore-aware code positioning on a heterogeneous dual-core processor consisting of a VLIW-based DSP core and a general-purpose core. The VLIW core is based on the HPL-PD 1.1 architecture [43], and the general-purpose core is similar to the Alpha 21264 processor [45]. More specifically, the simulation tools of Trimaran [24] and Chronos [46] (including SimpleScalar [47]) are extended to simulate this framework. The front end of Chronos compiles the other thread benchmark into

COFF format binary code by gcc compiler, which is targeted to SimpleScalar. By disassembling the binary code, the global CFG and related information of instructions are acquired by Chronos front end, hence helping static cache analysis. And a commercial ILP solver-CPLEX [48] is used to solve the ILP problem to obtain the estimated WCET.

Without losing generality, we assume a dual-core processor with two-levels cache memories. Each core has its own L1 instruction cache and L1 data cache, and both cores share the same L2 cache to utilize the aggregate L2 cache space. Note that multi-core processors can also use separated L2 caches to achieve better time predictability; however, a shared L2 cache has some important advantages such as fast data sharing, reduced cache-coherency complexity and false sharing and possibly superior cache performance [57]. To limit the scope of this study and to focus on instruction cache analysis, the L1 data cache of each core is assumed to be perfect. To compare the worst case performance with average case performance in the heterogeneous dual-core processor, the memory hierarchy of SimpleScalar simulator is integrated into Trimaran's memory hierarchy, and the core of SimpleScalar is linked to Trimaran's simulator by means of multi-thread programming. Therefore, an environment where two threads can run at the same time on different cores with a shared L2 cache has been simulated. The basic configuration of simulated hybrid dual-core processor is shown in Table 3.1.

In our experiments, we choose sixteen benchmarks from Mälardalen WCET benchmark suite [49], based on which we form eight benchmark pairs by selecting

Parameter	Dual-core Value	
Core	VLIW Core	General-Purpose Core
Datapath	4IFUs, 2FPUs, 2Ld/Sts, 1BrU 64 Registers	4 Issue, 64 Registers, 80-RUU, 40-LSQ
L1 I-cache	128 bytes, direct-map, 8 bytes block, 1 cycle latency	128 bytes, direct-map, 8 bytes block, 1 cycle latency
L1 D-cache	perfect	
L2 cache	2048 bytes, direct-map, 16 bytes block, 4 cycle latency	
Memory	unlimited, 100 cycle latency	

Table 3.1. Basic configuration of simulated heterogeneous dual-core processor.

Thread I					Thread II				
	Estimated WCET	Estimated L2 Miss Rate	Simulated WCET	Simulated L2 Miss Rate		Estimated WCET	Estimated L2 Miss Rate	Simulated WCET	Simulated L2 Miss Rate
Bs	3040	58.70%	2401	55.18%	Fft1	10677	30.69%	8988	26.97%
Cover	29987	18.97%	24918	16.30%	Ndes	367695	2.61%	332330	2.36%
Expint	10488	61.64%	8570	51.52%	Qsort	26793	18.81%	20208	16.35%
Fdct	16496	8.56%	13982	7.24%	Startup	11710	34.91%	9246	30.14%
Insertsort	5627	60.29%	4116	55.18%	Fibcall	3426	59.18%	2214	46.51%
Qurt	10375	31.06%	8127	25.23%	Crc	105904	3.50%	85327	3.22%
Sqrt	9042	60.64%	7030	54.39%	Minver	20798	30.46%	16843	27.64%
Ud	24175	18.75%	19615	16.71%	Biquad	7943	47.79%	6128	46.06%

Table 3.2. Estimated and simulated worst-case performance results of the baseline scheme.

a thread from each group as shown in Table 2. The performance results of WCO scheme, PFO scheme, and AFO scheme are compared with the baseline performance results in which no code positioning approach is applied.

3.5 EXPERIMENTAL RESULTS

3.5.1 Performance Results of WCO

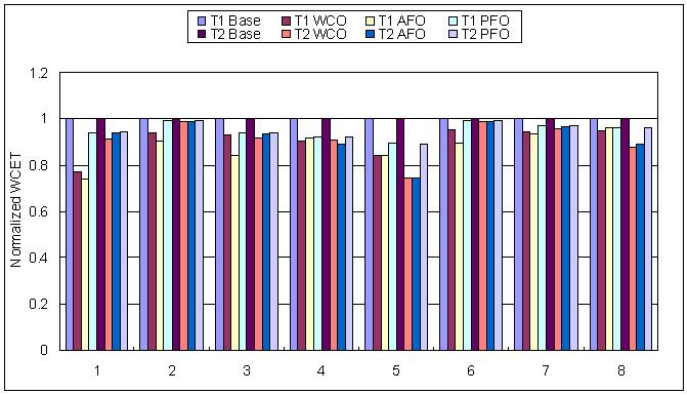
Figure 3.2 (a) shows the WCETs of eight benchmark pairs of the WCO scheme, which are normalized with respect to the results of the Baseline scheme. We can see that the WCO scheme can decrease the WCET for all the threads, because reducing the inter-thread L2 cache misses benefits both threads. The

percentages of WCET reduction for those eight benchmark pairs range from 1.14% to 15.85%, which depend on how much percentage the inter-thread L2 cache miss penalty takes in the WCET of each thread.

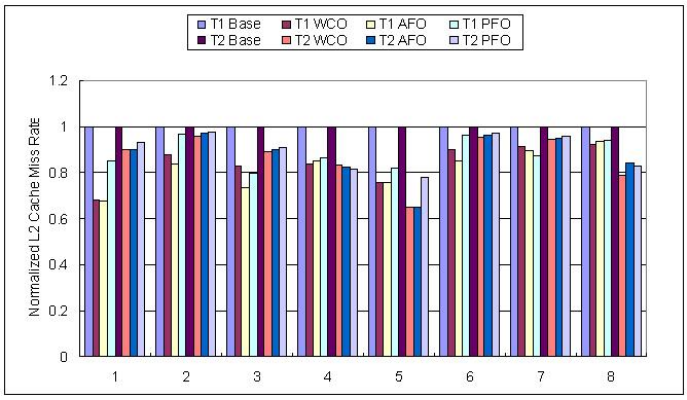
The variation of L2 cache miss rate of these benchmarks can be seen in Figure 3.2 (b). For the WCO scheme, it is likely that the thread with the longest WCET is not positioned, as the WCET of this thread is much larger than that of the other thread. In this case, the L2 cache miss rate of this thread with the longest WCET can be reduced more by the WCO scheme than both the PFO or AFO schemes, because no additional intra-thread L1 cache misses and intra-thread L2 cache misses will occur in this thread with the WCO scheme. For instance, in benchmark pair 3 both the WCET and L2 cache miss rate of benchmark `Qsort` in the WCO scheme are lower than those of the PFO and AFO schemes. In contrast, its counterpart benchmark `Expint` has higher L2 cache miss rate and larger WCET in the WCO scheme than those of the AFO scheme (which has better results than PFO). We also notice that for the benchmark pair 5, L2 miss rates and WCETs of both threads are adequately reduced by all three schemes, this is because the difference between the original WCETs of both benchmark pairs is relatively small and the L2 cache miss penalty takes a large fraction of their respective WCET.

3.5.2 Performance Results of PFO

The performance results of the PFO scheme are demonstrated in Figure 3.2 as well, which indicate that the PFO approach can reduce the WCETs of both threads within a benchmark pair by approximately equal percentages. For



(a) WCET



(b) L2 Cache Miss Rate

Figure 3.2. WCET and L2 cache miss rate of the WCO scheme, the AFO scheme and the PFO scheme which are normalized with respect to the Baseline scheme.

example, the difference of WCET reduction percentage for benchmark pair 2 consisting of `Cover` and `Ndes` is only 0.03%. Even for the worst case, the difference between WCET optimization percentage for benchmarks `Insertsort` and `Fibcall` is just 0.64%. On average, the variation of WCET optimization percentage for these eight benchmark pairs is only 0.29%.

However, we also find that the percentage of WCET reduction by PFO varies much across different benchmark pairs. For example, the WCET of the first

benchmark pair is reduced by more than 5%, while the percentages of WCET reduction for benchmark pair 2 and 6 are just about 1%. This is because the effect of the PFO approach on WCET reduction is mainly determined by two factors. First, to ensure fairness of WCET optimization, a wide discrepancy between the original WCETs of both threads limits the degree of WCET improvement for both benchmarks, for instance the benchmarks in pair 2 and pair 6. Second, the percentage of inter-thread L2 cache miss penalty in the original WCET is another important factor to determine the WCET enhancement through code positioning. The higher this percentage, the more space for potential WCET enhancement. The first factor also leads to another conclusion that the PFO approach is generally worse than other two approaches in terms of reducing the worst-case execution time (i.e., achieving "better" WCET), which can be observed in Figure 3.2 in case of both WCET and L2 cache miss rate. In other words, while the PFO approach can achieve fairness in terms of the percentage of WCET optimization for co-running threads, it indeed compromises the efficiency of WCET optimization as compared to WCO and AFO.

3.5.3 Performance Results of AFO

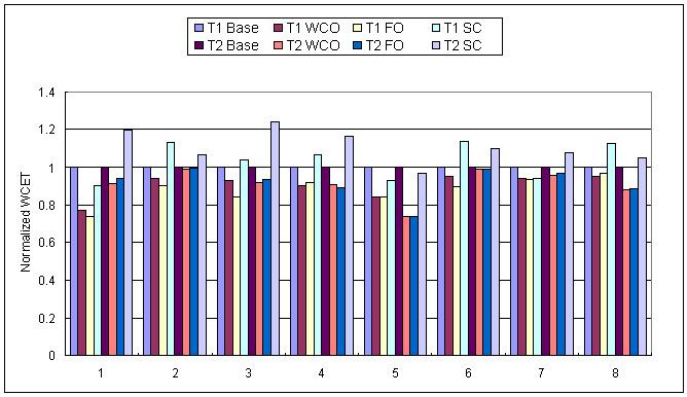
Figure 3.2 also illustrates normalized WCET and L2 cache miss rate in case of the AFO Scheme with respect to the Baseline Scheme. The AFO scheme can not only reduce the WCET and L2 cache miss rate for both threads in each benchmark pair, but also achieve the fairness in terms of the amount of WCET reduction. Specifically, the differences of reduced WCETs between both

benchmarks only range from 4 cycles to 120 cycles across all pairs. On average, the difference of WCET reduction is only about 80 cycles, which is less than the latency incurred from one L2 cache miss, indicating that the fairness in terms of the amount of WCET reduction between co-running threads is achieved.

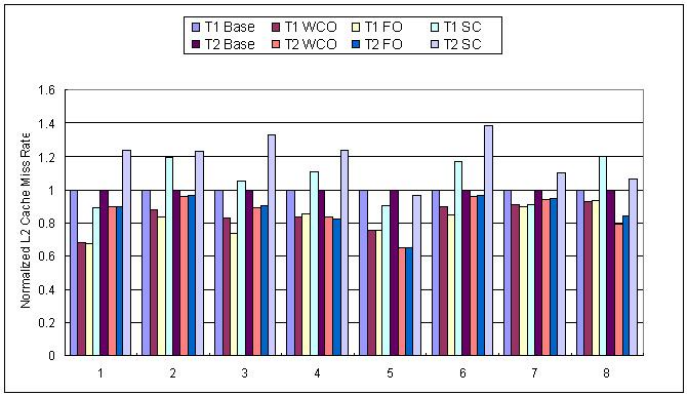
Interestingly, when we compare AFO with WCO (note that PFO in general is inferior to both AFO and WCO as aforementioned), we find that for all benchmark pairs, while WCO can decrease the WCET for one thread more than AFO, AFO can often reduce the WCET of the other thread (in the same pair) more than WCO. The reason is that the AFO approach de-positions instructions of one thread positioned by WCO and then positions the corresponding instructions of the other thread for reducing the inter-core L2 cache misses, which often leads to the increase of intra-core L2 cache miss on one thread, as well as the decrease of it on the other thread. As an example, in the first benchmark pair, the benchmark `Fft1` gets 2.14% improvement on WCET in the WCO scheme than in the AFO scheme; however, the WCET of `Bs` in the AFO scheme is about 3.29% better than that in the WCO scheme. Therefore, we believe AFO is comparable to WCO in terms of WCET optimization, while achieving fairness in terms of the amount of WCET reduction by considering both co-running threads.

3.5.4 Compare Code Positioning Schemes with Separated L2 Caches

In order to compare the performance of the proposed code positioning schemes with the technique of cache partitioning, in our experiments, the 2048 bytes L2 cache is separated in half and one thread can only access one of them to



(a) WCET



(b) L2 Cache Miss Rate

Figure 3.3. WCET and L2 cache miss rate of the WCO scheme, the AFO scheme (which is better than PFO) and the SC scheme, which are normalized with respect to the Baseline Scheme.

simulate a simple hardware-based cache partitioning (i.e. a separated L2 cache architecture), which is also called the SC scheme in this chapter. As shown in Figure 3.3 (a), for all benchmark pairs the WCETs of both or at least one of the WCO and AFO schemes are better than that of the SC Scheme. Even in some benchmarks, the performance of the SC Scheme is worse than that of the Baseline scheme, for instance `Cover` and `Qsort`. This is because although cache partitioning helps to reduce cache interferences between different threads, it may bring much more intra-thread cache conflicts as the actual cache mapping space is reduced by half. This is especially problematic if the code size of the working set exceeds the cache size, which is very likely in embedded processors due to the resource constraints. Therefore, for the benchmarks evaluated in this chapter, we believe that the code positioning approaches studied in this chapter are more effective than simply separating the L2 cache by half in improving the worst-case execution time for real-time tasks.

3.6 CONCLUSION

This chapter proposes novel code positioning approaches on multi-core platforms to co-optimize the worst-case performance for real-time threads running concurrently on a multi-core processor with a shared L2 cache. We have studied three different multicore-aware code positioning schemes to either maximally reduce the longest WCET or to ensure fairness of WCET enhancement among all co-running threads. Our experiments indicate that the WCO scheme can efficiently reduce the worst-case execution time for a single thread with the worst WCET,

and the AFO and PFO schemes can reduce the WCETs of co-running threads by approximately the same amount or percentage respectively. Also, the evaluation shows that the multicore-aware code positioning approaches are generally more effective than simply separating the L2 cache by half to reduce the WCET.

CHAPTER 4

ARCHITECTURAL TIME-PREDICTABILITY FACTOR (ATF): A NEW METRIC TO EVALUATE TIME PREDICTABILITY OF MICROPROCESSORS

4.1 CHAPTER OVERVIEW

As well known, accurately estimating the worst-case execution time (WCET) is crucial for hard real-time and safety-critical systems. However many traditional microprocessor architectural designs such as caches and branch prediction are aimed at improving the average-case performance, which unfortunately are harmful to time predictability [70, 71]. As a result, WCET analysis for modern processors has become very complex, if not impossible. The recent development of multithreaded and multicore architectures aggravates this problem. The resource contention in those architectures can adversely affect the execution time and further complicate WCET analysis. On the other hand, designing a microprocessor with high time predictability but low performance is likely to be useless. Therefore researchers have been studying time-predictable microprocessor design to reconcile time predictability and performance [70], with the goal to achieve better time predictability (or WCET analyzeability) while minimizing the impact on average-case performance.

Some designs of time-predictable processors have been proposed. Delvai et al. designed SPEAR (Scalable Processor for Embedded Applications in Real-Time

Environments), which employed a simple 3-stage pipeline and no cache memories [62]. Paolieri et al. [68] examined a time-predictable multicore architecture to support WCET analyzeability. Colnaric et al. [70] proposed a simple asymmetrical multiprocessor architecture for hard real-time applications, in which no dynamic architectural feature such as pipelines and caches was used. Yamasaki et al. [72] studied prioritized multithreaded processor through IPC control and prioritization. Edwards and Lee [63] proposed the precision timed (PRET) machine. Schoeberl [69] proposed a time-predictable Java processor. However, in all these studies, time predictability was not quantitatively evaluated, probably due to the lack of an effective and widely accepted metric when these studies were conducted.

Compared to the quantitative study of microprocessor design for improving the average-case performance, the time-predictable processor design so far has been a qualitative study and ad-hoc somehow. Because there is no well-defined metric to evaluate time predictability of processors, most prior work on time-predictable processor design either simply reported the worst-case performance through measurement or analysis, or qualitatively explained their designs were time-predictable by removing undesirable architectural features. The lack of a metric of time predictability thus not only prevents designers from understanding and comparing different time-predictable designs quantitatively, but also makes it difficult to make intelligent trade-offs between time predictability and average-case performance, which often conflict with each other. To make an analogy, it would be hard to imagine how much progress the computer architecture

community would have made without having a metric to quantitatively evaluate average-case performance!

Lately, defining a metric of time predictability has received considerable attention by the real-time and embedded computing community. To the best of our knowledge, Thiele et al. [70] defined time-predictability as the pessimism of WCET analysis and BCET analysis. Grund [73] defined time-predictability as the relation between BCET and WCET and argued that time predictability should be an inherent system property. Grund et al. [74] then proposed a template for predictability definitions and refined the definition into state-induced time predictability and input-induced time predictability. Kirner and Puschner [75] formalized a universal definition of time predictability by combining WCET analyzeability and the stability of the system. However, in all the above work except Grund et al. [73, 74], the calculation of time predictability is still dependent on the computation of WCET. Since the WCET estimation is usually pessimistic and there is no standard way to compute WCET (though different methods to derive WCET such as abstract interpretation and static cache simulation etc. do exist), any time predictability metric relying on WCET analysis is likely to be inaccurate and hard to be standardized in practice.

Moreover, in all the above works except Grund et al. [73, 74], the definition of time predictability does not separate the time variation caused by software and hardware, making it overly complicated to derive a time predictability metric that can effectively guide the architectural design for time predictability. While Grund

et al. [73, 74] proposed state-induced timing predictability (SIP) to separate timing uncertainty between hardware and software, the metric they proposed to evaluate SIP needs to exhaustively find out the maximum and minimum execution time of all different states, which may not be computationally feasible. In contrast, this chapter proposes a metric to efficiently assess architectural time predictability, and its effectiveness has been validated on a Very Long Instruction Word (VLIW) processor.

In this chapter, we make the following contributions to the time-predictable design of processors:

1. We introduce the concept of timing contract and architectural time predictability (ATP) to separate the timing unpredictability concern caused by hardware design from software, thus making it feasible to quantitatively assess and guide time-predictable architectural design.

2. We propose to use Architectural Time-predictability Factor (ATF) as a metric to quantitatively evaluate architectural time predictability of a processor, as well as architectural time predictability of various architectural and microarchitectural components of the processor.

3. We have evaluated the ATF of a VLIW processor as well as its microarchitectural components, including caches, parallel pipelines, branch predictor, speculative execution and the use of SPM. To the best of our knowledge, we are the first to use a quantitative metric to systematically evaluate the time predictability of a high-performance processor.

The remaining of this chapter is organized as follows. Section 4.2 introduces the concept of architectural time predictability. Section 4.3 defines the metric of architectural time-predictability factor. Section 4.4 qualitatively analyzes architectural time predictability of a VLIW processor. The evaluation methodology and the experimental results are presented in Section 4.5 and Section 4.6 respectively. Finally, the conclusions are made in Section 4.7.

4.2 ARCHITECTURAL TIME PREDICTABILITY

While static timing analysis aims at estimating the WCET safely and as close as possible to the actual WCET of a given processor, whether it is time-predictable or not; the goal of time-predictable architectural design is to design processor architectures so that their timing behavior can be precisely and efficiently predicted. To predict the timing behavior of a processor, we must have a desirable baseline timing behavior to compare with. This baseline time behavior is called the **timing contract** in this chapter, as it functions like a contract to guide the timing behavior of the actual execution. For example, the timing contract may specify how many cycles each instruction takes, in which order instructions can overlap their execution in the pipelines etc. If a processor is designed and implemented to fully enforce the timing contract, then it will be fully architecturally time-predictable. Therefore, we can then define architectural time predictability as the following.

Definition 1. Architectural Time Predictability: Given an architectural design of a processor, architectural time predictability indicates how close the actual timing behavior is to the baseline timing behavior specified in the timing contract of the processor.

Since not all the architectural designs are fully time-predictable, how do we specify the timing contract for an architectural component that is inherently not time-predictable? In that case, *the timing contract should specify the desired timing behavior while also ensuring high performance*. In other words, optimistic, not pessimistic assumption is preferred to establish an "ideal" baseline processor. For example, if a processor employs a cache memory, the desired timing behavior should be all cache hits, i.e. a perfect cache. While assuming all cache misses is still time-predictable, the performance will be too bad and hence is not desirable. On the other hand, when the timing behavior of an architectural component is totally time-predictable, no assumption, whether optimistic or not, should be made to objectively model the actual timing behavior. For example, if a processor employs a scratch-pad memory, then the latency of every load instruction is fixed and known (i.e. the data are either from the SPM or from the memory). Therefore, the timing contract of this processor should specify the latencies of all the loads without making further assumption.

It should be noted that ATP is independent of the timing uncertainty caused by software. If the input changes, a different path is exercised and the execution time can vary, but this processor can be still fully time-predictable if the execution

time exactly follows the timing contract (i.e. the timing variation caused by different inputs is the same for both the "ideal" processor specified in timing contract and the real processor). In other words, the goal of time-predictable processor design should not be to ensure the execution time is not varied or can be bounded with different inputs. Bounding the worst-case execution time with various inputs should be the business of WCET analysis, not the hardware design. However, a time-predictable processor can make WCET analysis in general and the low-level analysis in particular significantly easier as the impact of microarchitectural components (e.g. caching, branch prediction) on the execution time can be predicted or controlled.

4.3 ARCHITECTURAL TIME-PREDICTABILITY FACTOR

Built upon the definition of ATP, we propose to use **Architectural Time-predictability Factor** to quantitatively evaluate *architectural time predictability*. Given a processor P, an arbitrary real-time trace T, the actual dynamic execution time $D(P,T)$, and the statically predicted execution time based on the timing contract $S(P,T)$, ATF is defined as the following.

$$ATF(P,T) = \frac{D(P,T)}{S(P,T)} \tag{4.1}$$

It should be noted that here we evaluate ATP based on an arbitrary trace. Given different inputs, a real-time program may generate different traces, thus ATF for this program can be computed based on the ATFs of different traces, for example as an average or standard deviation of the ATFs for all the traces

evaluated. This is very similar to performance evaluation, in which we can get the execution time of each trace, and derive an average performance result across different traces to indicate the overall performance. Note that the execution time variation due to different inputs or traces are caused by software unpredictability. Techniques to analyze the worst-case program paths have been extensively studied in the literature of WCET analysis [71], which is complementary to the architectural time predictability studied in this chapter. To simplify discussion, we focus on studying ATF for an arbitrary trace in the rest of the chapter.

Given a trace T, $D(P,T)$ can be measured at runtime. Thus the remaining question is how we calculate $S(P,T)$. While $S(P,T)$ can be computed statically, it is quite different from static timing analysis, as we cannot require the processor to always produce the worst-case performance to make itself time-predictable. The $S(P,T)$ is statically computed according to the timing specification defined in the timing contract. Since the timing contract specifies the timing behavior of an architecture that is fully time-predictable, $S(P,T)$ should be independent of the machine states. For example, varied cache latencies are not allowed in a timing contract, as cache hits/misses depend on the history of cache accesses. In this chapter, we start the timing contract with a high-performance single-core processor with parallel pipelines, perfect caches, and no speculative execution so that the latency of each type of instructions, including loads and stores, can be statically specified.

The timing contract can be then exposed to the compiler to schedule

instructions, based on which $S(P,T)$ can be directly computed. Actually, modern optimizing compilers have already exploited the hardware timing information including latencies of various instructions to schedule instructions intelligently for maximizing resource utilization and attaining the best performance. Thus after compilation, not only the number of instructions but also the scheduling (i.e. static clock cycles) of each instruction can be known. Given a processor P and a trace T, the scheduling time of each instruction in T is usually assigned by the compiler based on a certain scope, e.g. a basic block or a superblock, based on which the statically predicted execution time of a trace can be easily calculated, which is simply called **static scheduling time** in this chapter. The details of computing *static scheduling time* for the processor we evaluate can be seen in Section 4.5.1.

Given a trace T, although the instructions of the trace are executed in a processor following the scheduling, their actual execution time may vary at runtime due to the performance-enhancing but non-time-predictable architectural features such as branch mis-predictions and cache misses. This is because the actual processor we implement may not have perfect pipelines, perfect branch prediction, and perfect caches etc. As a result, the actual execution time of a trace T on the given processor P is simply called **dynamic execution time** in this chapter, which can be directly measured on a real processor or a cycle-accurate simulator.

Thus given any processor P and any trace T, by applying Equation 4.1, ATF can be simply calculated in Equation 4.2. Typically, ATF should be no less than 1[1]. If *architectural time-predictability factor* is 1, it means the architecture is 100%

[1]ATF may be smaller than 1 in case that we are using a superscalar processor with out-of-order

time-predictable. Otherwise, the closer the ATF is to 1, the more time-predictable the architecture is.

$$ATF = \frac{dynamic\ exec\ time}{static\ sched\ time} \tag{4.2}$$

Why is ATF useful? Researchers in WCET analysis and computer architectures have already figured out certain hardware components such as caches, and branch prediction are not time predictable, so why do we need to use ATF? This is equivalent to say since caches are faster than main memory, a processor with a cache will definitely have better performance than a processor without a cache, thus there is no need to evaluate the actual performance of the processors with or without the cache. When designing a processor, a computer architect usually has multiple design objectives and constraints, including but not limited to average-case performance, energy dissipation, cost, compatibility, and time predictability for real-time systems, etc. It should be noted that while time predictability is surely an important design objective for real-time systems, computer architects are unlikely to only focus on achieving time predictability without considering other important design objectives such as average-case performance. Prior studies on time-predictable design are mostly qualitative in nature, which cannot tell quantitatively how good or how bad the time predictability is, or how much better the time predictability can be improved by

execution so that the dynamic execution sequence leads to less execution time than the **static scheduling time** predicted by the compiler. In this case, the ATF is less than 1, and the smaller the ATF, the more unpredictable the processor is.

applying a new design. With the availability of ATF, it becomes possible to quantitatively study the impact of architectural and microarchitectural design on time predictability, which can be used to make intelligent tradeoffs between time predictability and other design objectives. For example, cache locking is widely known to provide better time predictability for cache accesses. However, once a piece of data is locked into a particular cache block, that cache block cannot be dynamically reused to hold other data. As a result, the cache performance may degrade. For a processor that needs to balance time predictability and performance, designers might want to only lock a fraction of data or optimally reserve a fraction of cache space for locking while leaving the remaining cache lines for regular caching to achieving higher performance, which can be guided by ATF (for time predictability) and the execution time (for performance).

Is ATF larger than 1 useful? An ATF of 1 indicates perfect time predictability, which is an important design goal of hard real-time and safety-critical systems. However, there could be multiple architectural and microarchitectural designs that can achieve an ATF of 1, but with different impact on performance or energy. Thus, being able to evaluate the ATF of different architectural and microarchitectural design is crucial in this process. By comparison, without the ATF, it would be hard to validate the perfect time predictability, especially for complex processors. Moreover, today soft real-time systems, such as iphones or other handheld devices are widely and increasingly used in our society, for which the quality of service (QoS) is important.

Unfortunately, conventional architectural design such as multiprocessor present severe challenges when trying to provide even soft real-time guarantees. Thus, achieving an ATF close to 1, but not necessarily exact 1, could be beneficial for a wide variety of soft real-time systems, for which reducing the time variation, jitters and providing QoS are important.

Note that several prior studies [70, 75] used estimated WCET to compute time predictability. In this chapter, we use *static scheduling time* instead of WCET. The estimated WCETs often have different amount of overestimation, which can hardly make the time predictability evaluation accurate. In other words, the inaccuracy of WCET analysis should not be a reason to prevent us from deterministically evaluating time predictability. In contrast, *static scheduling time* is based on the compiler-generated schedule and the timing behavior specified in the timing contract, both of which are deterministic for a given trace. Also, since every program needs to be compiled before execution (the discussion on interpretation and dynamic compilation is out of the scope of this chapter), the methodology to estimate *static scheduling time* can be generally applied to different programs and various processors to provide a solid foundation for evaluating architectural time predictability.

4.4 QUALITATIVE ANALYSIS OF ATP ON A VLIW ARCHITECTURE

In this chapter, we validate the effectiveness of ATF on a VLIW architecture based on HPL-PD [23], which is a parametric processor architecture aiming at

improving instruction level parallelism (ILP) by adopting advanced compiler and architectural techniques. In a VLIW architecture, the compiler, not the hardware, is responsible for orchestrating the ILP of programs. To facilitate compiler scheduling, the VLIW architecture exposes as much hardware and timing information as possible to the compiler, such as the latency of each instruction, the number of functional units etc. Therefore, a VLIW processor is relatively more time-predictable than a superscalar processor, which dynamically schedules instructions by hardware. However, the HPL-PD based VLIW processor still has some architectural features that can compromise architectural time predictability as the following:

Branch architecture of HPL-PD not only replaces conventional branch instructions with a set of instructions to initiate a prefetch of the branch target early to minimize delays, but also uses a combination of bimodal branch predictor and global history with index sharing to predict the branch target dynamically [67]. In case of a branch mis-prediction of conditional branches, some stall time is added into the execution time at run-time.

Speculative execution in HPL-PD consists of control speculation and data speculation. Control speculation represents code motion across conditional branches. Data speculation is designed to increase the range of code motion for memory instructions. Speculative execution is generally safe but may lead to exceptions. If an exception is raised during the execution of a necessary speculated instruction, the recovery of the exception requires the re-execution of some

instructions, resulting in additional execution time. Also as exceptions can only be detected at run-time, speculative execution can possibly degrade architectural time predictability of the processor with the handling and recovery of any exception.

Cache memories of HPL-PD consist of first-level instruction and data caches and a second-level unified cache. Since the latency to access the memory hierarchy for an instruction depends on the result of accessing the caches (i.e. a hit or a miss), which can only be precisely known at run-time, the compiler always optimistically assumes a hit in the first-level cache for each memory access. Thus cache memories can lead to time unpredictability.

4.5 EVALUATION METHODOLOGY

We evaluate the ATP of the VLIW architecture based on Trimaran [24], which is an integrated compilation and performance monitoring infrastructure of VLIW architectures. We select 6 real-time benchmarks from Mälardalen WCET benchmark suit [64] and 4 benchmarks from MediaBench [66] for the experiments. The general information of these benchmarks is shown in Table 4.1.

The simulated processor is configured with 2 integer ALUs, 2 float ALUs, 1 branch unit, 1 load/store unit and 2-level caches. The 2-level caches consist of a level-1 instruction cache, a level-1 data cache and a level-2 unified cache. The parameters of the level-1 instruction cache are: size 512 bytes, block size 16 bytes, direct-mapped, miss penalty 7 cycles, LRU replacement policy; the parameters of the level-1 data cache include: size 1024 bytes, block size 32 bytes, direct-mapped, miss penalty 10 cycles and LRU replacement policy; and the parameters of the

benchmark	category	description	code size (bytes)	data size (bytes)
crc	real-time	cyclic redundancy check computation on 40 bytes of data	664	458
edn	real-time	finite impulse response (FIR) filter calculations	13504	3104
lms	real-time	lms adaptive signal enhancement	2136	1296
matmult	real-time	matrix multiplication of two 20 × 20 matrices	480	4828
ndes	real-time	complex embedded code	3580	986
statemate	real-time	automatically generated code	2476	498
cjpeg	mediabench	jpeg image compression	71468	135565
djpeg	mediabench	jpeg image decompression	70516	26508
mesamipmap	mediabench	OpenGL graphics clone: using mipmap quadrilateral	124892	39397
mesatexgen	mediabench	OpenGL graphics clone: texture mapping	180228	45074

Table 4.1. General information of all benchmarks

level-2 unified cache are: size 2048 bytes, block size 64 bytes, direct-mapped, miss penalty 100 cycles and LRU replacement policy. Note that due to the small sizes of the benchmarks, especially the real-time benchmarks, we use small cache configurations in our evaluation.

In a statically-scheduled VLIW processor, whenever there is a cache miss, the whole instruction pipeline will be stalled to wait until the data is returned. Therefore, the *dynamic execution time* of a trace running on the VLIW processor can be computed based on Equation 4.3, where *compute time* is the execution cycle through the pipeline, *cache stall time* is the stall cycle caused by cache accesses, and *branch stall time* is the stall cycle caused by branch mis-predictions.

$$\begin{aligned} \textit{dynamic exec time} = \textit{compute time} + \textit{cache stall time} \\ + \textit{branch stall time} \end{aligned} \tag{4.3}$$

In order to study not only the architectural time predictability of the processor but also that of each architectural component, we define the following

three component-level ATFs to indicate the ATP of speculative execution, caches, and branch prediction respectively. It should be noted that the component-level ATF just studies the effect of an unpredictable microarchitectural component on ATP, thus its value could be less than 1, and 0 indicates that this component does not have negative impact on architectural time predictability.

$$speculative\ ATF = \frac{(compute\ time - static\ sched\ time)}{static\ sched\ time} \tag{4.4}$$

$$cache\ ATF = \frac{cache\ stall\ time}{static\ sched\ time} \tag{4.5}$$

$$branch\ predictor\ ATF = \frac{branch\ stall\ time}{static\ sched\ time} \tag{4.6}$$

4.5.1 Static Scheduling Time Analysis

To quantitatively evaluate architectural time predictability of an architecture, static scheduling time of a trace must be analyzed accurately. In contrast, *dynamic execution time* can be easily obtained through simulation or measurement. In the HPL-PD architecture, the main idea of the *static scheduling time* analysis of a trace is to accumulate the *static scheduling time* of all basic blocks (BB)s according to the control flow and the scheduling time determined by intermediate representation(IR) of the program and the given input, which is described in Algorithm 5.

The algorithm begins with determining the weights (i.e. the execution

Algorithm 5 *Static Scheduling Time Analysis*

```
1: input: intermediate representation of the program and an input
2: output: static scheduling time of the trace
3: begin
4: Control_Flow_Profiling(IR, input)
5: Pipeline_Scheduling(IR)
6: for all BB do
7:     for each exit_edge of BB do
8:         if src_inst of current exit_edge is a real inst then
9:             BB_time+=(src_inst.sched_time+1) × exit_edge.weight
10:        else if src_inst of current exit_edge is a pseudo inst then
11:            BB_time+=src_inst.sched_time × exit_edge.weight
12:        end if
13:    end for
14:    if no exit_edge in BB then
15:        if last_inst of BB is a real inst then
16:            BB_time=(last_inst.sched_time+1) × BB.weight
17:        else if last_inst of BB is a pseudo inst then
18:            BB_time=last_inst.sched_time × BB.weight
19:        end if
20:    end if
21:    static_sched_time+=BB_time
22: end for
23: return static_sched_time
24: end
```

frequencies) of all BBs and its edges in the trace by control flow profiling based on the given input. Then the scheduling time of each instruction in the trace is calculated in the scope of the BB according to pipeline scheduling. Lines 7 to 13 calculate the *static scheduling time* of BBs with exit edges. If the source instruction of an exit edge is a real instruction, the *static scheduling time* of one execution of the BB related to this exit edge equals to the scheduling time of this instruction plus 1; otherwise, it only equals to the scheduling time of this instruction. The *static scheduling time* of the executions of the BB from an exit edge equals to the *static scheduling time* of one execution multiplied by the weight of this exit edge. Then the *static scheduling time* of the BB is the sum of the *static scheduling time* of the executions from all exit edges. In case of a BB without any exit edge as shown from Lines 14 to 20, the *static scheduling time* of one execution of the BB is calculated with the scheduling time of its last instruction. Then the *static scheduling time* of the BB equals to the *static scheduling time* of one execution multiplied by the weight of the BB. The algorithm is terminated when the *static scheduling time* of all BBs are accumulated, and its timing complexity is linear to the total number of the exit edges of the trace.

4.6 EXPERIMENTAL RESULTS

4.6.1 An Ideal VLIW Processor

First, we perform experiments on an ideal VLIW processor, which disables speculative execution and has a perfect cache and a perfect branch predictor. As shown in Table 4.2, architectural time-predictability factors of all benchmarks are

benchmark	static sched time	dynamic exec time	ATF
crc	20774	20774	1
edn	37655	37655	1
lms	260940	260940	1
matmult	81395	81395	1
ndes	46005	46005	1
statemate	1154	1154	1
cjpeg	12390627	12390627	1
djpeg	3839632	3839632	1
mesamipmap	25787205	25787205	1
mesatexgen	76954216	76954216	1

Table 4.2. ATF of all benchmarks in an ideal VLIW processor.

exactly 1. These data reveal that architectural time predictability of an ideal VLIW processor is perfect as one would expect.

4.6.2 A Realistic VLIW Processor

Figure 4.1 demonstrates ATFs of all benchmarks for a realistic VLIW processor. The bar of each benchmark in this figure consists of four components, including the normalized *static scheduling time*, speculative ATF, cache ATF and branch predictor ATF. The ATFs range from 1.26 to 11.18, and are 4.67 on average, indicating the realistic VLIW architecture is not fully time-predictable, which is consistent with our qualitative analysis in Section 4.4. We notice that the benchmark `statemate` has the worst ATF value. This is because it is a small benchmark that only takes 1154 computation cycles, so the memory stall time due to cache misses (mostly cold misses) becomes significantly larger than the *static*

benchmark	static sched time	compute time	cache stall time	BR stall time	speculative ATF	cache ATF	BR predictor ATF
crc	20774	20774	1619	3812	0	0.0779	0.1835
edn	37655	37655	54973	600	0	1.4599	0.0159
lms	260940	260940	336656	4956	0	1.2902	0.019
matmult	81395	81395	111573	1912	0	1.3708	0.0235
ndes	46005	46005	61850	3724	0	1.3444	0.0809
statemate	1154	1154	11694	52	0	10.1334	0.0451
cjpeg	12390627	12390627	38510460	551320	0	3.108	0.0445
djpeg	3839632	3839632	25169447	55784	0	6.5552	0.0145
mesamipmap	25787205	25787375	79860330	201828	0.00000659	3.0969	0.0078
mesatexgen	76954216	76954331	602816266	1072312	0.00000149	7.8334	0.0139

Table 4.3. Speculative ATFs, cache ATFs and branch predictor ATFs of a realistic VLIW processor.

scheduling time, leading to a very high ATF value.

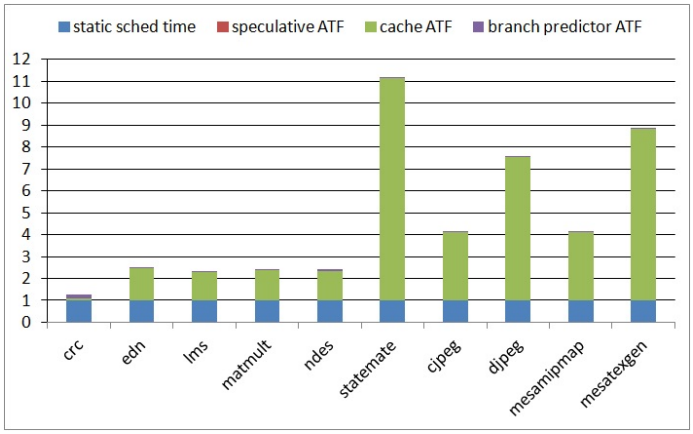


Figure 4.1. ATFs of all benchmarks in a realistic VLIW processor.

Table 4.3 gives the speculative ATFs, cache ATFs and branch predictor ATFs of the realistic VLIW processor. We observe that speculative ATFs are 0 for all benchmarks except *mesamipmap* and *mesatexgen*. This is due to the fact that only these two benchmarks have both instructions executed speculatively and the exceptions caught as shown in Table 4.4. On average, the speculative ATFs are still near 0, implying that while speculative execution can affect ATP, its impact is

benchmark	speculated inst	exceptions
crc	0	0
edn	0	0
lms	0	0
matmult	0	0
ndes	0	0
statemate	8	0
cjpeg	9462	0
djpeg	7588	0
mesamipmap	599172	10
mesatexgen	824499	10

Table 4.4. The number of speculated instructions and exceptions.

actually negligible for the VLIW processor we studied.

Table 4.3 also shows that branch predictor ATFs of all benchmarks range from 0.0078 to 0.1835, and are 0.0449 on average. The time variation between *static scheduling time* and *dynamic execution time* is due to the time of flushing the pipelines in case that the instructions on the mis-predicted paths are executed before the branch targets are known. Although the combined branch predictor is used in the VLIW processor, branch mis-predictions still occur and lead to the stall time. As shown in Table 4.5, branch stall time of each benchmark is proportional to the number of mis-predictions, which means ATP of the branch predictor can be improved by increasing the accuracy of the branch prediction. However, branch prediction only degrades the ATP of the processor by a comparatively small degree, because branch stall time is a relatively insignificant portion of the total *dynamic execution time*.

benchmark	branch inst	branch stall time	mis-prediction
crc	5553	3812	953
edn	4121	600	150
lms	28537	4956	1239
matmult	9707	1912	478
ndes	6209	3724	931
statemate	59	52	13
cjpeg	2160542	551320	137830
djpeg	197424	55784	13946
mesamipmap	3563318	201828	50457
mesatexgen	6787772	1072312	268078

Table 4.5. The number of branch instructions and the branch mis-predictions of all benchmarks

Additionally, cache ATFs of all benchmarks range from 0.0779 to 10.1334, and are 3.627 on average as shown in Table 4.3, which means time variation from memory hierarchy is not predictable. Because cache ATF is about 77% of ATF for all benchmarks on average, architectural time predictability of the VLIW architecture in study is mostly affected by time predictability of memory hierarchy.

4.6.3 Impact of The Number of Integer ALUs

The number of ALUs in a processor is another important factor that can affect ILP and the average-case performance. However, its impact on time predictability is not clear. Since the arithmetic instructions of the benchmarks in our experiments are mainly integer instructions, we perform some sensitive experiments on the number of integer ALUs (IALUs), which ranges from 1, 2 to 4.

As expected, increasing the number of IALUs reduces the dynamic execution

cycles of each benchmark as shown in Table 4.6. However, it does not imply that the time predictability will also become better. Actually as shown in Figure 4.2, ATF of each benchmark is increased with more integer ALUs, indicating worse time predictability. This is because with a larger number of IALUs, the compiler can also schedule more operations per cycle, leading to less *static scheduling time*. Interestingly, we found the reduction of *static scheduling time* is more than the *dynamic execution time*. The reason is that in a realistic HPL-PD processor, cache misses or branch misprediction can have greater impact on performance with more operations scheduled per cycle. However, this does not mean that changing the number of IALUs is inherently not time-predictable.

To verify our hypothesis mentioned above, we also conduct experiments with 1, 2 and 4 IALUs on the ideal VLIW processor. We find that the ATF is always 1 regardless of the number of IALUs and the *dynamic execution time* is reduced with the increase of IALUs. Therefore, changing the number of IALUs (or generally the functional units) itself should not affect the time predictability; however, due to its interaction with other time-unpredictable architectural components such as caches and branch predictors, the architectural time predictability of the processor could be affected.

4.6.4 Scratchpad Memory

Scratchpad memories (SPMs) [61] are used in embedded processors to improve time predictability and power efficiency. In a scratchpad memory system, the mapping of program and data elements is performed either by the user or by

benchmark	1 I-ALU	2 I-ALU	4 I-ALU
crc	33480	26205	26000
edn	101131	93228	89507
lms	605272	602552	591197
matmult	197997	194880	185282
ndes	125812	111579	108881
statemate	13076	12900	12512
cjpeg	53068587	51452407	50467769
djpeg	33063023	29064863	28794668
mesamipmap	113238593	105849533	103897385
mesatexgen	685328738	680842909	676586630

Table 4.6. The *dynamic execution time* with the number of integer ALUs varying from 1, 2 to 4.

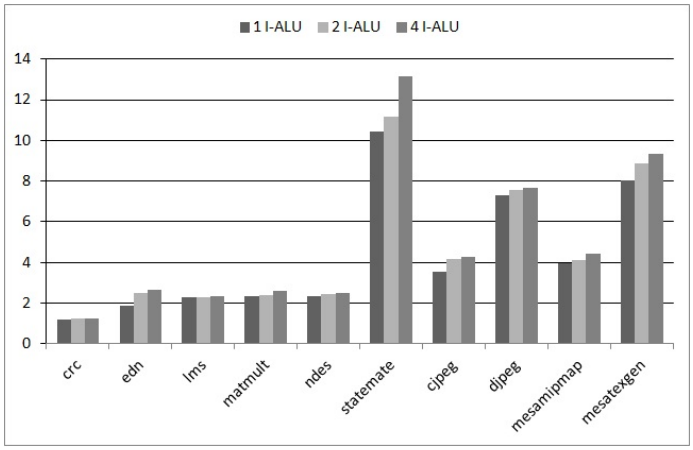


Figure 4.2. The ATF with the number of integer ALUs ranging from 1, 2 to 4.

the compiler using a suitable algorithm, resulting in predictable memory access time. In order to evaluate the effect of SPMs on ATP, we replace the 2-level caches in the processor with corresponding 2-level SPMs [65] including: a level-1 instruction SPM, a level-1 data SPM and a level-2 unified SPM. The size and the latency of each SPM are the same as the corresponding cache described in Section 4.5.

In our SPM allocation method, both instructions and data of a trace are assigned to SPMs by the compiler in the descending order of the number of accesses until all SPMs are filled. The assignment starts from the level-1 SPMs. For the level-2 unified SPM, a fair assignment policy is adopted for simplicity, that is a half of the level-2 unified SPM is assigned to instructions and data respectively. The same policy based on the number of accesses is used for the level-2 SPM allocation as well.

As shown in Figure 4.3, ATFs of the processor with SPMs are much less than those of the processor with caches, indicating using SPMs can significantly enhance architectural time predictability. On average ATF of the processor with SPMs is 1.02 and it is 3.65 times less than that of the processor with caches. Because the memory stall time of a trace depends on the assignment of instructions and data on SPMs, it can be calculated precisely after the compilation and included in the *static scheduling time* for the processor with SPMs. However, the ATF of the processor with SPMs is still not 1, which is mainly caused by timing variation due to branch mis-prediction and mis-speculative execution with exceptions.

However, *dynamic execution times* of all benchmarks except `crc` and `statemate` are increased by using SPMs instead of caches, as shown in Table 4.7. This is because the assignment of instructions and data in SPMs is fixed and no space in SPMs can be used by multiple instructions/data, leading to longer memory stall time in case the total size of instructions and data is larger than the size of SPMs or caches. In contrast, the caches can dynamically reuse the limited space to get better memory performance. For `crc` and `statemate` however, due to their small code and data footprints, all their instructions and data can be totally assigned into SPMs, hence leading to better performance. In summary, compared to caches, SPMs can significantly improve ATP; however, they can possibly degrade the average-case performance of the processor if the SPM space is not used efficiently[2].

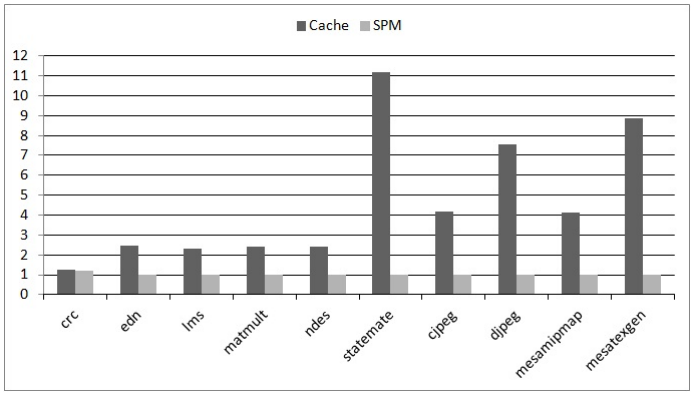


Figure 4.3. ATFs of a processor with SPMs compared with ATFs of a processor with caches.

[2]Please note this is just based on the SPM implemented in our experiments, which is not an optimal SPM allocation method. Also, dynamic SPM allocation may improve performance by reusing the SPM space more efficiently; however, this is out of the scope of this chapter.

benchmark	cache	spm	spm/cache ratio
crc	26205	24600	93.88%
edn	93228	795655	853.45%
lms	602552	611683	101.52%
matmult	194880	1050607	539.10%
ndes	111579	313057	280.57%
statemate	12900	9618	74.56%
cjpeg	51452407	510420442	992.02%
djpeg	29064863	225820936	776.96%
mesamipmap	105849533	680353365	642.76%
mesatexgen	680842909	7072252472	1038.75%

Table 4.7. Dynamic execution times of all benchmarks in a processor with SPMs compared with those in a processor with caches.

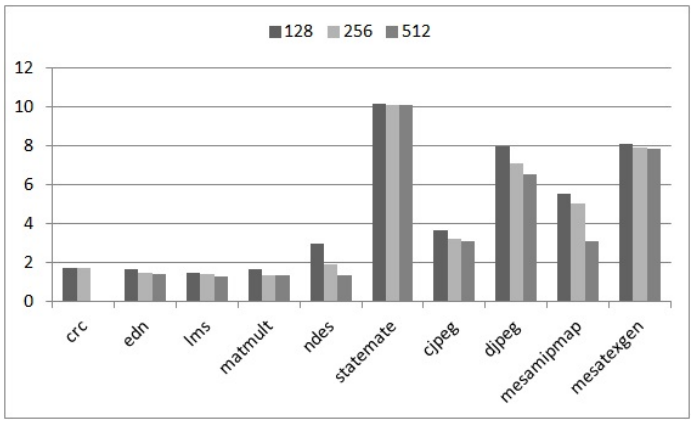
4.6.5 Sensitive Experiments of Cache Size

We have also performed sensitivity analysis to examine the impact of different cache sizes on cache ATF. In sensitive experiments of the L1 instruction cache, the size of the L1 instruction cache ranges from 128 bytes, 256 bytes, to 512 bytes; while the size of the L1 data cache is fixed to be 1024 bytes, and the size of the L2 unified cache is fixed to be 2048 bytes (other parameters are the same as those described in Section 4.5). As shown in Figure 4.4(a), cache ATF of each benchmark except `statemate` is decreased with the increase of the L1 instruction cache size, because cache stall time is reduced with the decrease of the L1 instruction cache miss rates as depicted in Figure 4.4(b). For `statemate`, it is a very small benchmark whose instruction accesses suffer mostly from cold misses, thus increasing the instruction cache size does not lead to noticeable reduction on

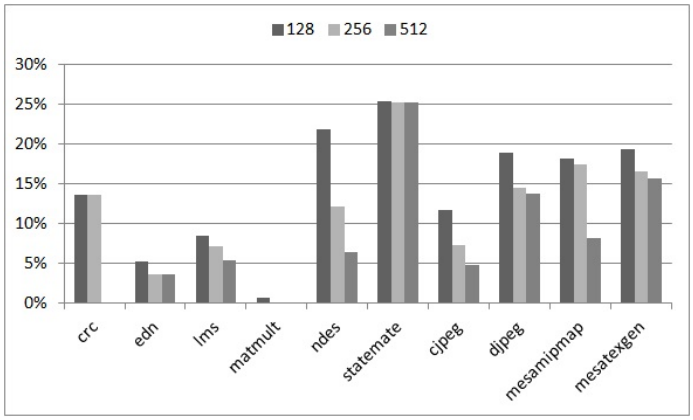
the instruction cache misses and *dynamic execution time*. Consequently, the impact on ATF is insignificant.

We also observe that both `crc` and `matmult` have small code size. Thus when the instruction cache size increases to 512 bytes and 256 bytes respectively, their instruction cache miss rates drop to very small values (i.e. 0.12% and 0.3%). The cache ATF of `crc` decreases to 7.8% when the instruction cache size is 512 bytes, because `crc` also has small data footprint and the cache stall cycles are dominated by instruction cache misses. By comparison, `matmult` has larger data footprint, thus its cache ATF decreases when the instruction cache size is increased to 256 bytes but stays almost the same when the instruction cache size is increased to 512 bytes.

In sensitive experiments of the L1 data cache, the size of L1 data cache ranges from 256 bytes, 512 bytes, to 1024 bytes; the size of L1 instruction cache is always 512 bytes; and the size of L2 unified cache is always 4096 bytes (other parameters are the same as those described in Section 4.5). As demonstrated in Figure 4.5(a), cache ATF of each benchmark is decreased with the increase of the L1 data cache size, because cache stall time is reduced with the decrease of the L1 data cache miss rate as shown in Figure 4.5(b). We notice that while `crc` is a small benchmark with small data footprint, most of its data accesses are cold misses, thus increasing the L1 data cache size does not significantly reduce its data cache miss rate. Since the cache stall cycles are only a small fraction of the total execution cycles for `crc`, its ATF is very small as compared to other benchmarks.

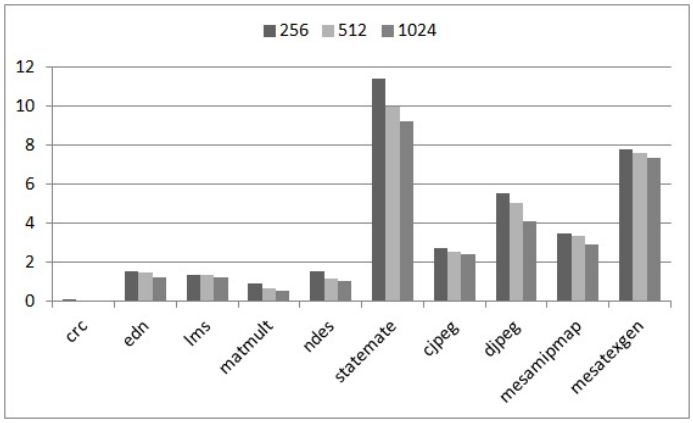


(a) cache ATF

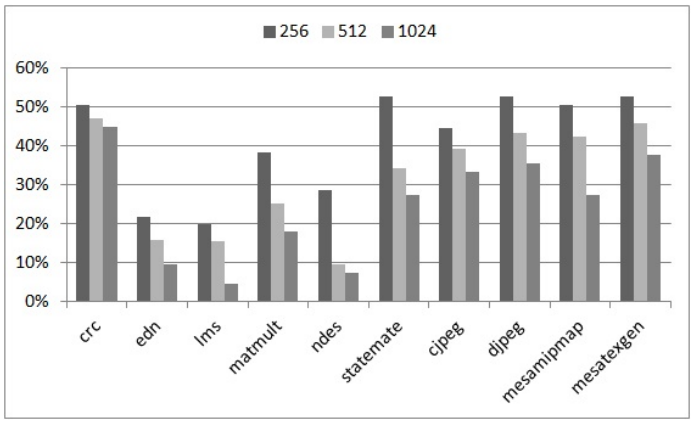


(b) L1 instruction cache miss rate

Figure 4.4. Cache ATF and L1 instruction cache miss rate sensitive to the size of L1 instruction cache.



(a) cache ATF

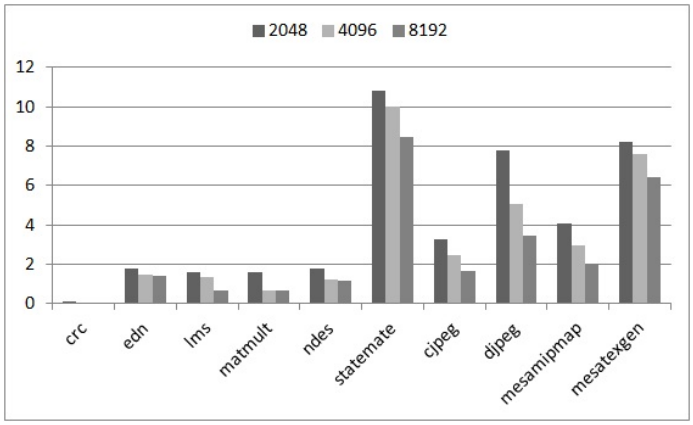


(b) L1 data cache miss rate

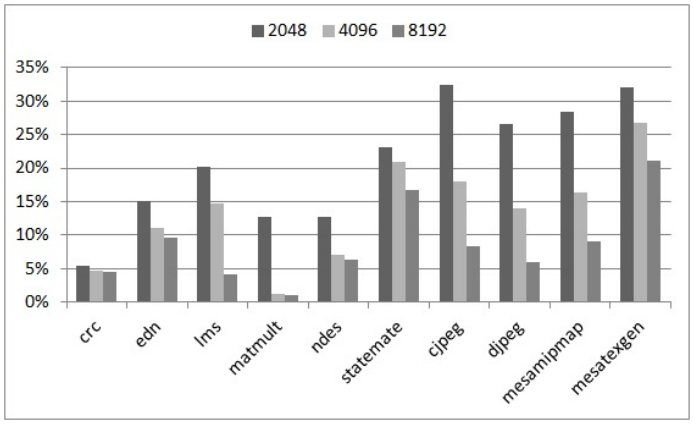
Figure 4.5. Cache ATF and L1 data cache miss rate sensitive to the size of L1 data cache.

Specifically, the ATF is 9.3%, 7.5%, and 6.5% when the L1 data cache size is 256 bytes, 512 bytes, and 1024 bytes respectively.

In sensitive experiments of the L2 unified cache, the size of L2 unified cache ranges from 2048 bytes, 4096 bytes, to 8192 bytes; the size of both L1 instruction and data caches are fixed to be 512 bytes (other parameters are the same as those described in Section 4.5). As shown in Figure 4.6(a), cache ATF of each benchmark is decreased with larger L2 unified cache sizes, because cache stall time



(a) cache ATF



(b) L2 unified cache miss rate

Figure 4.6. Cache ATF and L2 unified cache miss rate sensitive to the size of L2 unified cache.

is reduced with the decrease of the L2 unified cache miss rate as depicted in Figure 4.6(b). Overall we find increasing the L2 cache size is most effective at improving ATF due to its effectiveness on reducing the cache stall time. However, increasing the cache size also adds hardware cost and may increase the cache access latency, therefore there is a trade-off designers should make.

4.7 CONCLUSION

In order to guide the time-predictable architectural design for enhancing time predictability, we present the concept of architectural time predictability to separate the timing uncertainty concern of hardware design from software. Then we propose a new metric named architectural time-predictability factor to quantitatively evaluate architectural time predictability. The availability of such a metric allows computer architects to quantitatively evaluate the impact of different architectural/microarchitectural techniques on time predictability of processors, in addition to other important design objectives such as performance and energy dissipation, thus enabling them to make intelligent tradeoffs among time predictability, performance and energy consumption, which often conflict with each other. Without a metric like this, making quantitative tradeoffs will be impossible, and design for time predictability is at most an art, not science.

Our evaluation on a VLIW processor demonstrates that the proposed metric can effectively assess architectural time predictability of the processor, as well as architectural time predictability of various architectural and microarchitectural components. More specifically, our evaluation indicates that while speculative execution, branch prediction and cache memories can all affect architectural time predictability, caches have the most significant impact on ATP of the VLIW processor we studied. Moreover, our experiments quantitatively show that using large caches can improve both performance and time predictability; increasing the number of functional units can improve performance but degrade time

predictability (though not inherently); and using SPMs instead of caches can increase time predictability but may degrade performance.

CHAPTER 5

HYBRID ON-CHIP MEMORY ARCHITECTURE

5.1 CHAPTER OVERVIEW

Traditionally, computer architectural design has mainly focused on improving the average-case performance (or simply called performance in this paper) or energy efficiency recently. As a result, some performance-enhancing architectural techniques such as caches and branch prediction are harmful to time predictability [70, 71], which is crucial for hard real-time and safety-critical systems. With the recent trend on multi-threaded and multicore architectures, the worst-case execution time (WCET) analysis [71] for those architectures becomes much more complicated, making it extremely hard if not impossible to accurately derive the WCET. Moreover, today soft real-time systems, such as iphones or other handheld devices have been widely and rapidly used in our society, for which the quality of service (QoS) is important. Unfortunately, conventional architectural design such as multiprocessor present severe challenges when trying to provide even soft real-time guarantees [76]. Therefore, it becomes increasingly important to improve time predictability of computing while keeping high performance.

Cache memories have been widely used in modern processors to effectively bridge the speed gap between the fast processor and the slow memory to achieve good average-case performance. However, the cache performance is heavily dependent on the history of memory accesses and the cache placement and

replacement algorithms, making it hard to accurately predict the worst-case execution time. Scratch-Pad Memory (SPM) [61] is an alternative on-chip memory to the cache, which has been increasingly used in embedded processors such as ARMv6 and Motorola MCORE due to its energy and area efficiency. In a processor with SPM, the mapping of program and data elements into the SPM can be performed either by the user or the compiler, resulting in statically predictable memory access time. However, the performance of SPMs is generally not as good as that of caches because caches can dynamically reuse their space efficiently to benefit more instructions and data, especially for applications with large instruction and data footprints. To summarize, processors that employ caches or SPMs alone can only benefit either the average-case performance or the time predictability, not both.

This chapter proposes seven hybrid on-chip memory architectures (also simply called hybrid architectures in this paper) to combine caches and SPMs to reconcile performance and time predictability. Specifically, instead of using a single cache (or SPM) with size N, we propose to use a SPM with size M(M¡=N) and a cache with size N-Min parallel. Such a hybrid SPM-cache architecture can be used to store either instructions or data, which is called Instruction Hybrid (IH) architecture or Data Hybrid (DH) architecture respectively. We use the compiler to allocate a fraction of instructions or data to the SPM until it is full, while the rest of instructions or data are stored in main memory, which can exploit the cache for enhancing performance.

We three main contributions. First, we propose hybrid SPM-cache architectures that can leverage SPMs to achieve time predictability while allowing the use of caches for instructions and/or data not stored in the SPMs to improve the average-case performance. Second, we have systematically explored seven different hybrid on-chip memory architectures to understand how to make best use of both caches and SPMs to store instructions and data for balancing performance and time predictability. Third, while most prior works indicate performance and time predictability generally conflict with each other, this research shows that it is possible to exploit hybrid architectures intelligently for improving both time predictability and performance.

We have implemented and evaluated all the proposed seven hybrid architectures on a cycle-accurate simulator. The assessment of time predictability is based on the proposed metric of Architectural Time-predictability Factor (ATF) in Chapter 4. Our evaluation indicates that the hybrid architectures can generally make better tradeoffs between performance and time predictability than either caches only or SPMs only, which are actually two extremes of the spectrum of hybrid on-chip memory architectures. Among all the hybrid architectures, we find that using the hybrid SPM-cache for both instructions and data can optimally benefit both real-time programs with superior time predictability and non-real-time programs with higher performance.

The remaining of the paper is organized as the follows. Section 5.2 discusses the motivation for this work. Section 5.3 introduces a variety of hybrid on-chip

memory architectures by combining caches and SPMs. Section 5.4 describes our evaluation methodology, and Section 5.5 gives the experimental results. The related work is discussed in Section 5.6. Finally, we make conclusions in Section 5.7.

5.2 MOTIVATION

To quantitatively study performance in terms of the total number of execution cycles and time predictability in terms of ATF, we first evaluate two baseline architectures, including a pure cache, and a pure SPM based architectures, which are shown in Figure 5.1. The first baseline architecture employs only an instruction cache (IC) and a data cache (DC), and thus is referred as the IC-DC architecture in this paper. The other baseline architecture contains only an instruction SPM (IS) and a data SPM(DS), and is called the IS-DS architecture accordingly. The experiments are conducted by following the evaluation methodology and configurations presented in Section 5.

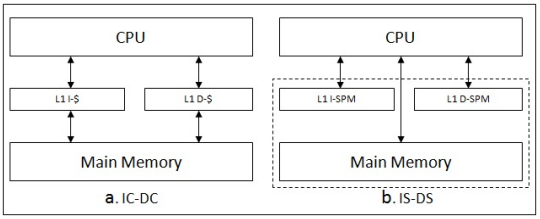
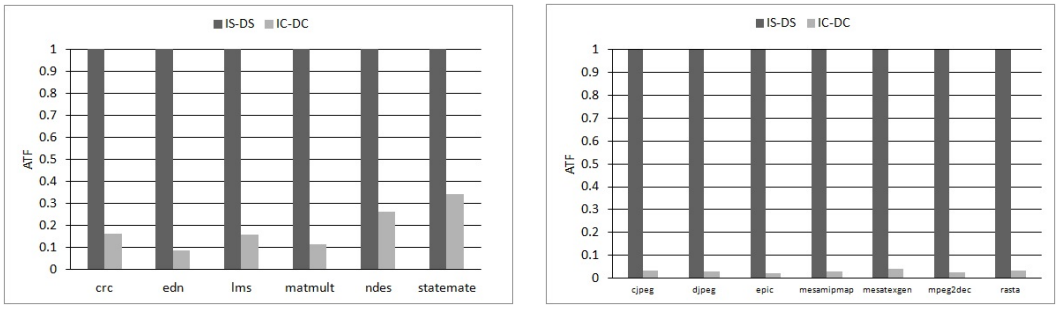


Figure 5.1. Two baseline architectures of the on-chip memories studied.

Figure 5.2 compares the ATFs of all the benchmarks between the IC-DC architecture and the IS-DS architecture. The ATF of each benchmark on the IS-DS architecture equals to 1, while the ATF of each benchmark on the IC-DC

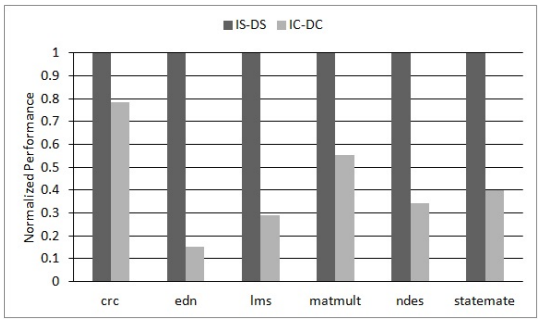


(a) the ATF of real-time benchmarks (b) the ATF of media benchmarks

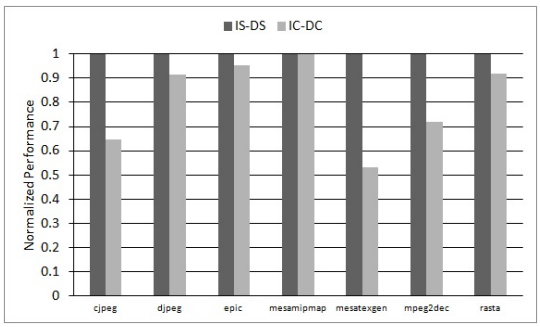
Figure 5.2. The comparison of the ATF of all benchmarks between IC-DC and IS-DS architectures.

architecture is much less than 1. On average, the ATF of the real-time benchmarks [64] on the IC-DC architecture is only 0.188, and the ATF of mediabenchs [66] on the IC-DC architecture is only 0.029. These very low ATF values quantitatively confirm our hypothesis that the IC-DC architecture has very bad time predictability, and thus is not desirable for real-time computing.

Figure 5.3 gives the performance (i.e. the total number of execution cycles) of the IC-DC and the IS-DS architectures, which is normalized to the performance of the IS-DS architecture. Except for mesamipmap, whose data footprint is small and can mostly fit in the 16K data SPM, the IC-DC architecture leads to much less execution cycles (i.e. better performance) than the IS-DS architecture for all other benchmarks. On average, the number of execution cycles of the real-time benchmarks on the IC-DC architecture is only 42% of that of the IS-DS architecture, and the number of execution cycles of the mediabenchs on the IC-DC architecture is about 20% less than that of the IS-DS architecture, indicating that the IS-DS architecture generally has inferior performance.



(a) the performance of real-time benchmarks



(b) the performance of media benchmarks

Figure 5.3. The comparison of the performance of all benchmarks between IC-DC and IS-DS architectures.

In summary, neither the IC-DC nor the IS-DS architecture can achieve both good time predictability and high performance. Therefore, it is desirable to explore new on-chip memory architectures to make better tradeoffs between time predictability and performance.

5.3 HYBRID ON-CHIP MEMORY ARCHITECTURES

5.3.1 Hybrid SPM-cache Architectures

Since both caches and SPMs have their own advantages and disadvantages, it would be desirable to combine their advantages while avoiding their respective disadvantages. To achieve this goal, we propose a hybrid SPM-cache architecture by tightly coupling caches and SPMs cooperatively to achieve both high performance and time predictability, which can potentially benefit a wide variety of applications, including both real-time and non-real-time (or general-purpose) programs. Figure 5.4 shows three such hybrid SPM-cache architectures. The first architecture has a hybrid SPM-cache for storing instructions and a regular data

cache, which is named as the IH-DC architecture; the second one has a regular instruction cache and a hybrid SPM-cache for data, which is called the IC-DH architecture; and the third one employs hybrid SPM-caches for both instruction and data, which is referred as the IH-DH architecture.

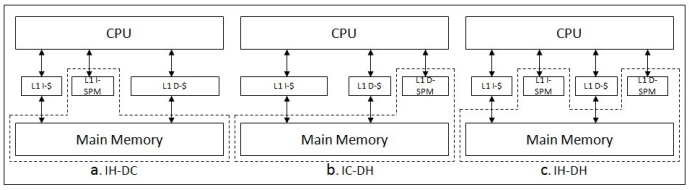


Figure 5.4. Three hybrid architectures of the on-chip memories proposed.

In the proposed hybrid SPM-cache architectures, the SPMs are used to achieve time predictability, while the cache is used to boost average-case performance by adapting to runtime behavior of instructions and data that are not stored into the SPMs. The WCET analysis of hybrid SPM-cache architectures consists of two parts: the analysis of the SPM and the analysis of the cache. The former is very simple and straightforward; whereas the latter can become very complicated or overestimated for traditional caches but can become simpler or even optional for the hybrid SPM-cache. First, while traditional timing analysis techniques for caches [71] can still be applied to the cache in the hybrid SPM-cache, they do not have to be applied if the complexity of analysis becomes prohibitive. The reason is that in the hybrid SPM-cache, the most frequently used instructions and data are already saved in the SPMs to guarantee a decent worst-case execution time. Second, due to the same reason, even if traditional

timing analysis methods for caches are applied to get tighter WCET, the overestimation is expected to be much smaller than that of a pure cache based architecture. In addition, since the cache in the hybrid SPM-cache is usually much smaller than a regular cache, the number of states needed to model and the complexity of analysis are expected to be reduced significantly even if the traditional timing analysis method needs to be used.

In the hybrid architecture, the SPM is mapped into an address space disjoint from the off-chip main memory, but is connected to the same address and data buses as the cache. The instructions and/or data are assigned to the SPMs by software. Thus after SPM allocation, an instruction or data can be stored either in the SPM or in the off-chip memory. In the latter case, the instruction or data are accessed by the processor through the small instruction or data cache within the hybrid SPM-cache architecture, which can exploit the temporal and spatial locality dynamically for improving the average-case performance.

There have been many studies on efficient SPM allocation algorithms to improve either the average-case performance [78, 79, 80] or WCET [81, 82, 83]. In this chapter, we develop a simple static SPM allocation algorithm for both instructions and data by exploiting profiling information. More advanced SPM allocation algorithms can be used to exploit the SPM space more efficiently, which, however, is not the focus of this paper. In our SPM allocation method, the instructions are assigned into the instruction SPM in the unit of a basic block. All the basic blocks are sorted in the descending order based on their weights (i.e. the

number of times each basic block is accessed). If a basic block has a larger weight and the total size of the instructions in it is less or equal to the remaining size of the instruction SPM, its instructions will be assigned into the instruction SPM earlier. Similarly the data objects are assigned into the data SPM by the compiler in the descending order of the number of accesses, subject to the capacity of the data SPM. The SPM allocation ends until the instruction/data SPM is filled fully.

Algorithm 6 describes our SPM allocation method in detail, where the memory object is a basic block in case of the instruction SPM and is a data object in case of the data SPM. The algorithm ends when all the memory objects are checked or there is no available space left in the SPM. The computational complexity is linear to the number of the memory objects to be checked.

Algorithm 6 *SPM Allocation*

```
1: input: the list of the memory objects MOList and the empty SPM
2: output: the SPM with the memory objects assigned
3: begin
4: Sort_By_Frequency_Descending_Order(MOList)
5: MO = MOList.head
6: while MO is not null do
7:     if SPM.avail_size ¿ 0 then
8:         if MO.size ¡= SPM.avail_size then
9:             assign MO into SPM
10:            SPM.avail_size = SPM.avail_size - MO.size
11:        end if
12:        MO = MO.next
13:    else
14:        break
15:    end if
16: end while
17: end
```

It is worthy to note that the hardware cost of the proposed hybrid SPM-cache is expected to be low. Since a SPM is usually more energy and area efficient than a cache with the same size [61], the hardware cost of a hybrid SPM-cache is unlikely to be more than that of a regular cache with equivalent capacity, which is especially important for embedded systems. However, since SPM is used in the hybrid SPM-cache, programs need to be compiled or recompiled for SPM allocation. This may be a disadvantage for legacy code. However, as multicore has become the mainstream, and many programs need to be recompiled anyway for achieving higher thread-level parallelism, we believe this trend provides excellent opportunities to explore new on-chip memory architectures such as the hybrid SPM-caches proposed in this chapter.

5.3.2 Design Space Exploration

In addition to the proposed hybrid SPM-caches, there are also other types of hybrid on-chip memory architectures, for example using a cache for instructions and a SPM for data. Generally, depending on the use of a cache, a SPM, or a hybrid SPM-cache for storing either instructions or data, there are totally 9 different combinations as shown in Table 5.1. Among these 9 different architectures, two are homogeneous: IC-DC is the traditional cache only architecture, and IS-DS is the traditional SPM only architecture, both of which can serve as the baselines for comparing performance and time predictability respectively. Figure 5.4 has illustrated three hybrid SPM-cache architectures, and the other four hybrid architectures include Instruction Cache and Data SPM

	D-Cache	**D-Hybrid**	**D-SPM**
I-Cache	IC-DC	IC-DH	IC-DS
I-Hybrid	IH-DC	IH-DH	IH-DS
I-SPM	IS-DC	IS-DH	IS-DS

Table 5.1. All the hybrid on-chip memories studied.

(IC-DS), Instruction SPM and Data Cache (IS-DC), Instruction Hybrid and Data SPM (IH-DS), and Instruction SPM and Data Hybrid (IS-DH). The first two use a cache or a SPM to store either instructions or data but not both. The latter two involve the hybrid SPM-cache, in addition to a regular SPM, to store either instructions or data. The performance and time predictability in terms of ATF of all these nine architectures will be comparatively evaluated in Section 6.

5.4 EVALUATION METHODOLOGY

5.4.1 Simulation and Benchmarks

We use Trimaran compiler/simulator framework [24] to evaluate the hybrid on-chip memory architectures on a VLIW processor. The baseline processor has 2 integer ALUs, 2 float ALUs, 1 branch predictor, 1 load/store unit, and 1-level on-chip memory. To focus on studying the impact of on-chip memories on ATP and performance, we assume perfect branch prediction and no speculative execution.

We randomly select 6 real-time benchmarks from Mlardalen WCET benchmark suit [64] and 7 media benchmarks from MediaBench benchmark suit [66] (also referred as media benchmarks in this chapter) for the experiments. The

benchmark	category	description	code size (bytes)	data size (bytes)
crc	real-time	cyclic redundancy check computation on 40 bytes of data	520	158
edn	real-time	finite impulse response (FIR) filter calculations	13504	3104
lms	real-time	lms adaptive signal enhancement	2072	1296
matmult	real-time	matrix multiplication of two 20 × 20 matrices	480	4828
ndes	real-time	complex embedded code	3452	986
statemate	real-time	automatically generated code	4112	498
cjpeg	mediabench	jpeg image compression	50960	135565
djpeg	mediabench	jpeg image decompression	46060	26508
epic	mediabench	an image compression program	19608	329611
mesamipmap	mediabench	OpenGL graphics clone: using mipmap quadrilateral	71240	39397
mesatexgen	mediabench	OpenGL graphics clone: texture mapping	98792	45074
mpeg2dec	mediabench	MPEG digital compressed format decoding	30252	389669
rasta	mediabench	A program for speech recognition	55384	132369

Table 5.2. General information of all benchmarks

	instruction cache						data cache					
size(bytes)	128		64		32		128		64		32	
	#A	#M	#A	#M	#A	#M	#A	#M	#A	#M	#A	#M
crc	30415	4142	6861	1386	6097	1720	685	95	101	60	57	40
edn	70525	3671	41758	8402	39962	10743	11352	1813	7863	1609	6767	1884
lms	252778	21369	184438	42065	152963	43970	38176	14368	24665	15473	19841	12254
matmult	123608	909	14288	2642	6928	2678	24009	12823	23689	14423	23529	18018
ndes	66988	14553	60336	16236	56240	16129	10262	4083	5516	3816	4828	4040
statemate	1560	396	1528	388	1516	386	391	238	227	148	188	132

Table 5.3. The number of accesses (#A) and the number of misses (#M) in both instruction caches and data caches of different sizes for the real-time benchmarks.

latter are used to represent non-real-time applications. The salient characteristics of all benchmarks are shown in Table 5.2. In addition, Table 5.3, Table 5.4 and Table 5.5 give the number of accesses and misses in both instruction caches and data caches of different sizes for the real-time benchmarks and the media benchmarks respectively.

Since the real-time benchmarks have much smaller memory footprints, we

	instruction cache					
size(bytes)	16K		8K		4K	
	#A	#M	#A	#M	#A	#M
cjpeg	17475531	2589	284792	3850	106668	4122
djpeg	5709754	1607	54825	1488	19020	1349
epic	131296282	625	4638	379	1830	244
mesamipmap	36812743	1190	30327	1352	24184	1338
mesatexgen	106062777	1160922	20699173	618371	12184154	1106090
mpeg2dec	162191325	60543	1488492	25108	368499	6294
rasta	13384811	46455	483925	37754	280168	20449

Table 5.4. The number of accesses (#A) and the number of misses (#M) in instruction caches of different sizes for the media benchmarks.

	data cache					
size(bytes)	16K		8K		4K	
	#A	#M	#A	#M	#A	#M
cjpeg	2291367	30422	542829	17501	383155	35985
djpeg	1093027	4106	69261	742	28722	854
epic	5991299	108704	378564	10200	282307	56149
mesamipmap	5621187	29037	53424	485	13680	157
mesatexgen	15993504	78973	398005	4682	36129	506
mpeg2dec	22324579	78859	1944936	40806	1900438	51975
rasta	1868454	90678	399831	28756	224512	15853

Table 5.5. The number of accesses (#A) and the number of misses (#M) in data caches of different sizes for the media benchmarks.

choose to use two different on-chip memory configurations. In the experiments of the real-time benchmarks, the sizes of the on-chip memories are 128 bytes for both instructions and data respectively. The parameters of the caches include: 16B block size, direct-mapped, and LRU replacement policy. A cache hit takes 1 cycle and a memory access takes 20 cycles. In the experiments of the media benchmarks, the size of the on-chip memories are 16K bytes for both instructions and data respectively. The parameters of the cache include: 32B block size, 4-way set-associative and LRU replacement policy.

In all the experiments on hybrid SPM-caches, we try two different partitions of the cache and the SPM, while keeping the total hybrid SPM-cache size fixed. For an N-byte hybrid SPM-cache i with the partition of a M-byte cache and an (NM)- byte SPM, we refer it as the i-M scheme. For example, for a 16K IH-DC architecture with a 4KB instruction cache and a 12K instruction SPM, it is denoted as IH-DC-4K in this chapter.

5.4.2 Static Execution Time Analysis

The main idea of computing the static scheduling time of a program is to accumulate the static scheduling times of all basic blocks generated by the instruction scheduler of the compiler. The computation of the static scheduling time is described in Algorithm 5 in Chapter 4.

After the SPM allocation is performed by the compiler, the instructions and the data which are in the SPMs become known, so the number of accesses to the instructions and data not in the SPMs can be computed, which is denoted as A.

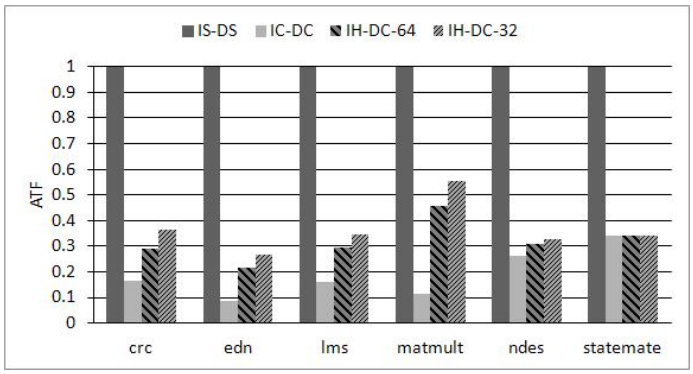
As the timing contract assumes that all the accesses to the instruction and the data not in the SPMs take memory access latency L, the static execution time can be computed by Equation 5.1

$$\begin{aligned} static\ exec\ time = static\ sched\ time \\ +A \times L \end{aligned} \tag{5.1}$$

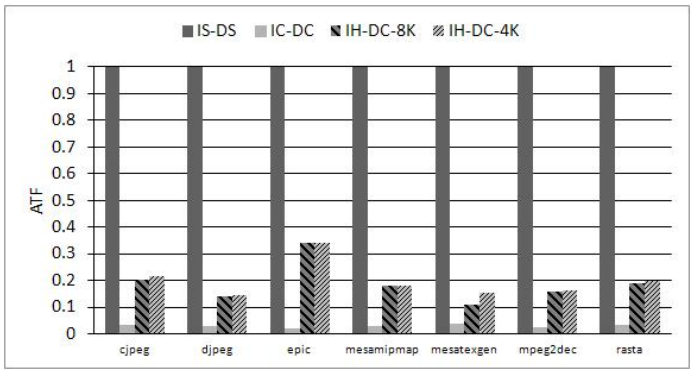
5.5 EXPERIMENTAL RESULTS

5.5.1 IH-DC Architecture

Our first experiment studies the time predictability of the IH-DC architecture, and the results are shown in Figure 5.5. For all the benchmarks, the ATFs of the IH-DC architecture with different cache/SPM partitions are better than those of the IC-DC architecture, but are less than those of the IS-DS architecture. This indicates that the IH-DC architecture can improve time predictability over the IC-DC architecture. Also we observe that for IH-DC with a fixed size, increasing the portion of SPM size leads to higher ATF. For example, IH-DC-32 has better ATFs than IH-DC-64 for real-time benchmarks, and IH-DC-4K has higher ATFs than IH-DC-8k for media benchmarks, implying that the ATP can be improved by increasing allocation of on-chip memory size to the SPM. However, the IH-DC architecture still has less ATF than the IS-DS architecture, because the IH-DC architecture still contains a small instruction cache and a regular data cache, both of which have varied access latencies that can harm time predictability.



(a) the ATF of IH-DC for real-time benchmarks



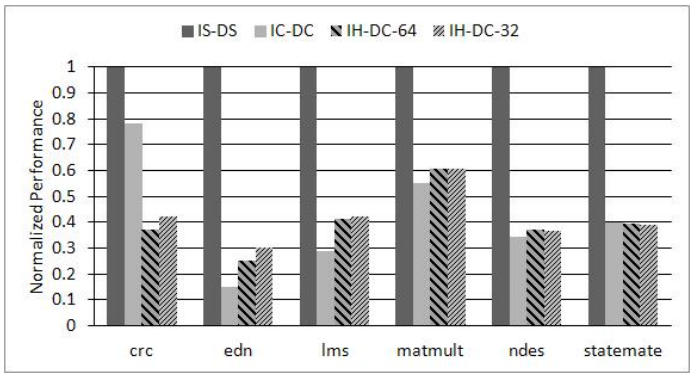
(b) the ATF of IH-DC for media benchmarks

Figure 5.5. The comparison of the ATF of all benchmarks between IH-DC, IC-DC and IS-DS architectures.

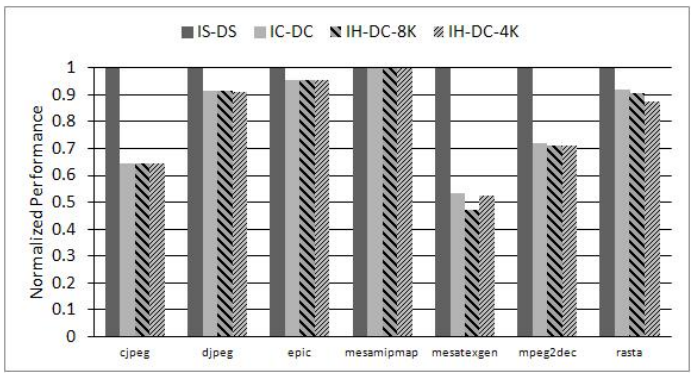
Figure 5.6 compares the performance of the IH-DC architecture with the IS-DS and IC-DC architectures, which is normalized to the performance of the IS-DS architecture. For most benchmarks, we observe that the IH-DC architecture has much better performance than the IS-DS architecture, and has performance comparable to the IC-DC architecture for most benchmarks. Interestingly, we find for several benchmarks such as crc and statemate from the real-time benchmarks and mesatexgen and rasta from the mediabench, the IH-DC architecture actually leads to better performance than the IC-DC architecture. For crc, mesatexgen, and rasta the instruction cache misses in IC-DC architecture are mainly conflict misses caused by the instructions in the basic blocks with the highest frequencies, which are assigned into the SPM first. Consequently, the number of cache misses can be significantly reduced by using the IH-DC architecture, as shown in Tables 3 and 4 respectively. For statemate however, it only contains a few loops; so the number of accesses to each instruction is small, and the cache misses are dominated by cold misses. By directly accessing some instructions from the SPM, the IH-DC architecture can reduce the cost of cold misses, leading to better performance.

5.5.2 IC-DH Architecture

Our second set of experiments evaluate the time predictability and performance of the IC-DH architecture. Figure 5.7 compares the ATFs among the IC-DH architecture with two different partitions, the IS-DS, and the IC-DC architectures. We observe that the IC-DH architecture can achieve better ATFs than the IC-DC architecture for all the benchmarks except statemate; however, it



(a) the performance of IH-DC for real-time benchmarks

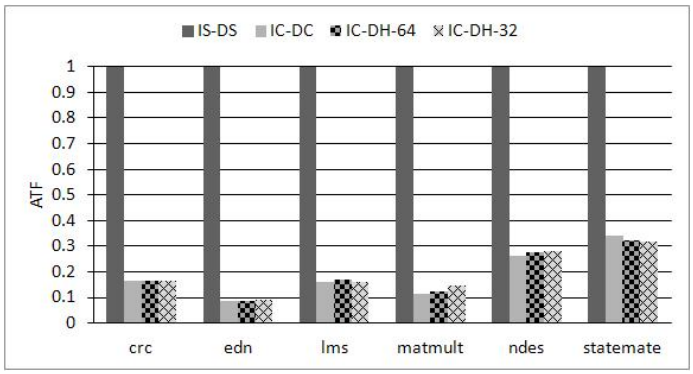


(b) the performance of IH-DC for media benchmarks

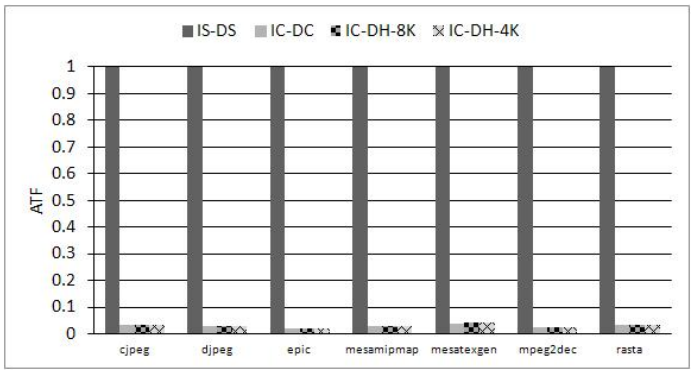
Figure 5.6. The comparison of the performance of all benchmarks between IH-DC, IC-DC and IS-DS architectures.

has much less ATF than the IS-DS architecture. This indicates that while IC-DH can improve time predictability over the pure cache based architecture, the improvement is very limited. The reason is mainly because in the IC-DH architecture, a regular instruction cache is still used, which can have larger adverse impact on time predictability than a regular data cache, as instructions are accessed in every clock cycle. For statemate, we find the percentage of the static execution time reduced by accessing a fraction of data from the data SPM is less than the percentage of the dynamic execution time reduced by having less data cache misses, resulting in a lower ATF.

Figure 5.8 presents the performance of the IC-DH architecture, and the IS-DS and IC-DC architectures, which is normalized to the performance of the IS-DS architecture. As we can see, the IC-DH architecture has better performance than the IS-DS architecture for all the benchmarks. For real-time benchmarks, the performance of 4 out of 6 benchmarks in IC-DH architecture is comparable to that of the IC-DC architecture, including crc, edn, lms, and ndes. One special case is statemate, whose execution time is reduced by 5% and 6% in IC-DH-64 and IC-DH-32 respectively, compared to the performance of the IC-DC architecture. This is because the number of data cache misses is reduced from 238, 148, to 132 with the decrease of the data cache size from 128, 64, to 32, as more data can be stored in the data SPM. However, for matmult, the performance of IC-DH-64 and IC-DH-32 are about 5% and 16% worse than that of the IC-DC architecture respectively. The reason is that the number of data accesses to the data cache is



(a) the ATF of IC-DH for real-time benchmarks



(b) the ATF of IC-DH for media benchmarks

Figure 5.7. The Comparison of the ATF of All Benchmarks Between IC-DH, IC-DC and IS-DS Architectures.

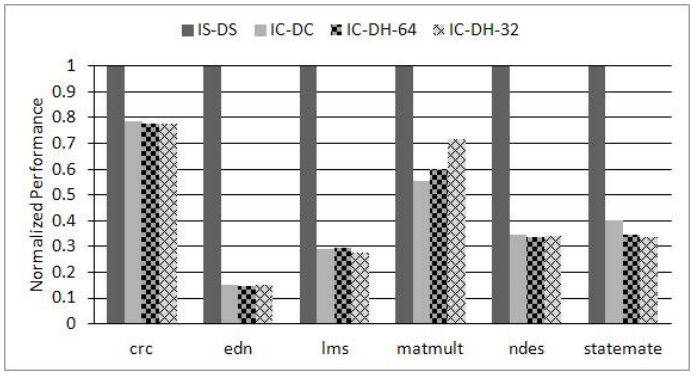
only slightly reduced by increasing the size of the data SPM as shown in Table 5.3, while the number of data cache misses is significantly increased with the decrease of the data cache size.

For media benchmarks, we observe the IC-DH architecture leads to better performance than the IC-DC architecture for all the benchmarks except cjpeg. Even for cjpeg, the performance in IC-DH-8K is about 1.2% better than that of the IC-DC architecture. This indicates that with proper allocation of space between the data cache and the data SPM, IC-DH architecture can achieve performance superior to the IC-DC architecture for media benchmarks.

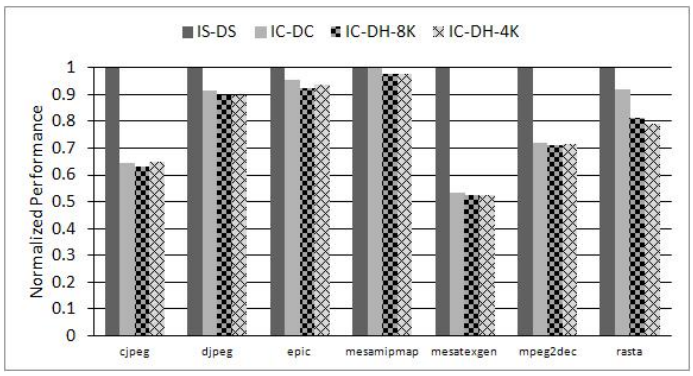
5.5.3 IH-DH Architecture

Figure 5.9 compares the ATFs of the IH-DH architecture with the IS-DS, IC-DC, IH-DC, and IC-DH architectures. For all the benchmarks except the real-time benchmark statemate, the ATFs of the IH-DH architecture are larger than those of the IC-DC, IH-DC, and IC-DH architectures but are less than those of the IS-DS architecture. This is because in IH-DH, both instructions and data can exploit the hybrid SPM-caches to enhance time predictability. Actually for some media benchmarks such as mesamipmap, the ATF of the IH-DH architecture is very close to that of the baseline architecture IS-DS. Also, we observe that for the IH-DH architecture, the ATF of each benchmark in the partition with a larger SPM is higher than that of the partition with a smaller SPM, because the SPM access latency is deterministic while the cache access latency can be varied.

Figure 5.10 compares the performance of the IH-DH architecture with the

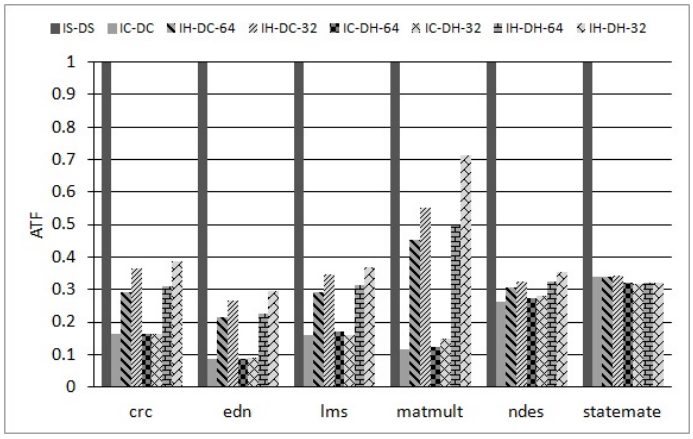


(a) the performance of IC-DH for real-time benchmarks

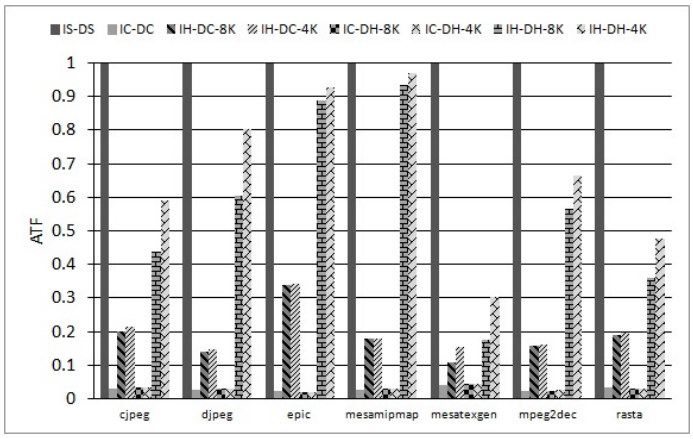


(b) the performance of IC-DH for media benchmarks

Figure 5.8. The comparison of the performance of all benchmarks between IC-DH, IC-DC and IS-DS architectures.



(a) the ATF of IH-DH for real-time benchmarks



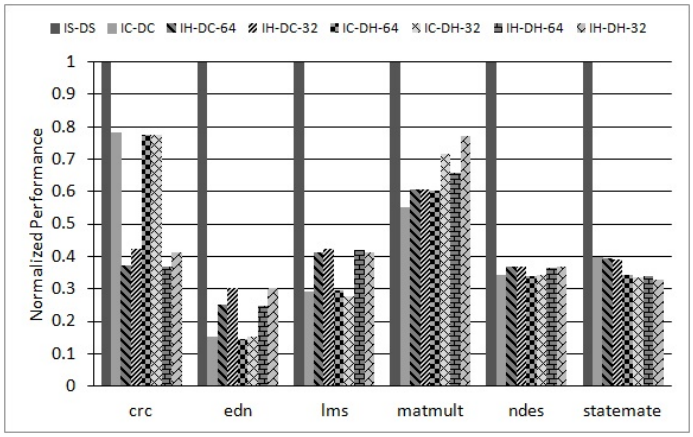
(b) the ATF of IH-DH for media benchmarks

Figure 5.9. The comparison of ATFs among IH-DH and IS-DS, IC-DC, IH-DC, and IC-DH architectures.

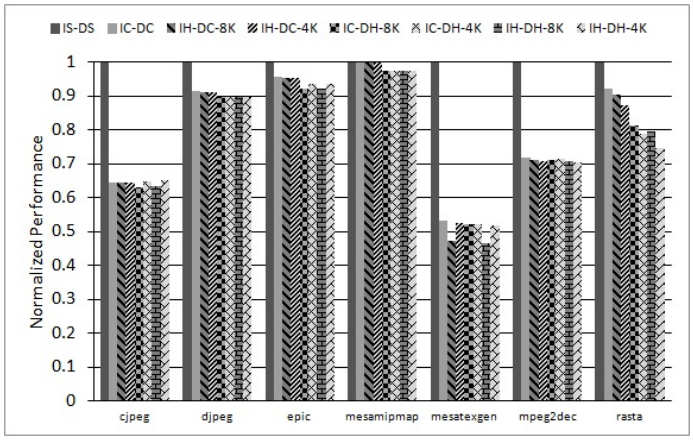
IS-DS, IC-DC, IH-DC, and IC-DH architectures, which is normalized to the performance of the IS-DS architecture. We find that the IH-DH architecture outperforms the IS-DS architecture for all the benchmarks, and has performance comparable or better than both the IH-DC and IC-DH architectures for most benchmarks. For 4 out of 7 real-time benchmarks, including edn, lms, matmult and ndes, the performance of the IH-DH architecture is worse than that of IC-DC architecture. This is because the instruction accesses of these benchmarks are very sensitive to the size of the instruction cache, thus decreasing the size of the instruction cache leads to significantly more cache misses than the number of cache misses reduced by increasing the size of the SPM, as can be seen from Table 3. However, we also find that the IH-DH architecture can result in higher performance than the IC-DC architecture for many other benchmarks, including crc, statemate from real-time benchmarks and epic, mesamipmap, mesatexgen, and rasta from mediabench. On average, the performance of IH-DH is 1.9% better than that of the IC-DC architecture for real-time benchmarks, and is 4% better for media benchmarks, indicating that IH-DH can enhance both time predictability and performance on average.

5.5.4 Comparing All 9 Architectures

In addition to the three hybrid on-chip memory architectures we have studied, we have also evaluated four other hybrid architectures involving a pure SPM, including IC-DS, IH-DS, IS-DC, and IS-DH. Figure 5.11 compares the ATFs of these 7 hybrid on-chip memory architectures with the two baseline architectures



(a) the performance of IH-DH for real-time benchmarks



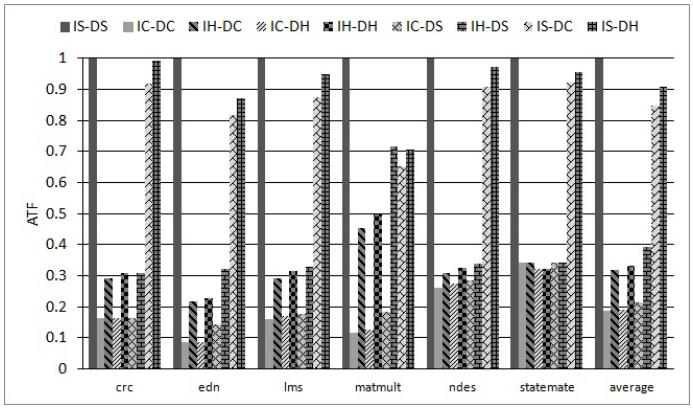
(b) the performance of IH-DH for media benchmarks

Figure 5.10. The comparison of performance among IH-DH and IS-DS, IC-DC, IH-DC, and IC-DH architectures, which is normalized to the performance of IS-DS architecture.

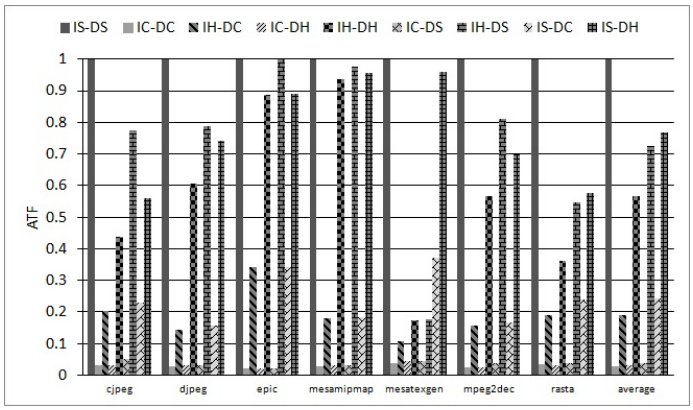
IS-DS and IC-DC. For each hybrid SPM-cache architecture with two different partitions between the cache and the SPM, we present the best ATF results of different partitions. In general, the ATFs of all proposed hybrid on-chip memory architectures are larger than that of the IC-DC architecture, indicating that all the hybrid on-chip memory architectures can achieve better time predictability than the pure cache based architecture. Particularly, we find IS-DH achieves the highest ATF among all the hybrid on-chip memory architectures, because instruction access latency can significantly affect the architectural time predictability.

Also we observe that the ATF in IH-DH architecture for each benchmark is larger than those in the hybrid on-chip memory architectures without using any pure SPM. In some cases, the IH-DH architecture can achieve time predictability close to those hybrid on-chip memory architectures with the pure SPM. For example, the ATF of epic with the IH-DH architecture is 0.888, while it is 0.889 for the IS-DH architecture.

Figure 5.12 compares the performance of these 9 architectures, which is normalized to the performance of the IS-DS architecture. Similarly, for each hybrid SPM-cache architecture with two different partitions between the cache and the SPM, we present the best performance results of different partitions. In general, the performance of all proposed hybrid on-chip memory architectures are better than that of the IS-DS architecture. The performance of the hybrid on-chip memory architectures without any pure SPM is close to the performance of the IC-DC architecture. Actually, some of them, such as IH-DH can even achieve



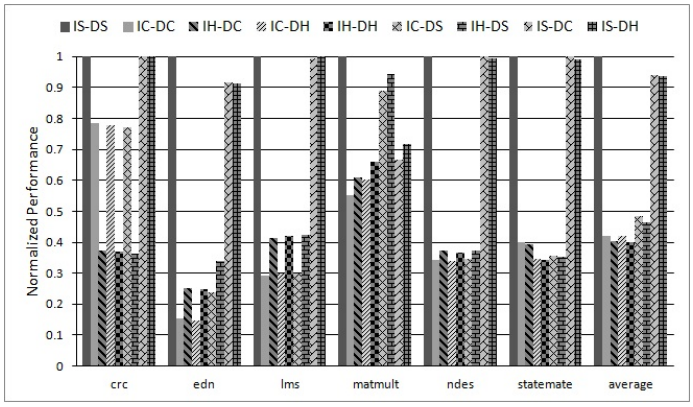
(a) the ATF for real-time benchmarks



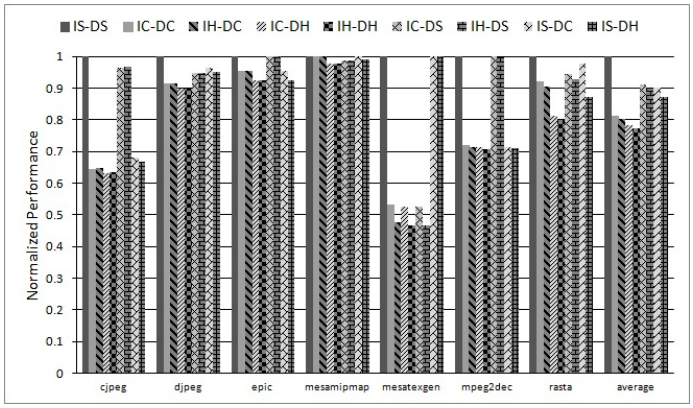
(b) the ATF for media benchmarks

Figure 5.11. The comparison of the ATFs among all 9 architectures.

better performance than the IC-DC architecture as aforementioned.



(a) the performance for real-time benchmarks



(b) the performance for media benchmarks

Figure 5.12. The comparison of performance among all 9 architectures, which is normalized with the performance of the IS-DS architecture.

Between IH-DC and IC-DH, we find the IH-DC architecture always leads to higher ATF for all the benchmarks, while the IC-DH architecture can attain better performance for some benchmarks. Compared to IC-DH, on average, the ATF of the IH-DC architecture is 12.7% higher for real-time benchmarks and 15.7% higher for media benchmarks. In terms of the averaged performance, IH-DC is 1.6% better than IC-DH for real-time benchmarks; but IC-DH is 1.7% better than

IH-DC for media benchmarks. Due to the large improvement of ATF and comparable performance, it seems IH-DC is superior to IC-DH. This again indicates the importance of putting instructions into a more deterministic on-chip memory to enhance the overall time predictability.

Between IC-DS and IS-DC, we find that IS-DC has much better ATF, comparable or better performance for media benchmarks, though worse performance for most real-time benchmarks. Overall it seems combining a pure instruction (data) cache with a pure data (instruction) SPM is not a very good idea. This is because IS-DH can achieve better ATF than IS-DC, and IH-DS can achieve better performance than IC-DS.

Overall, we find that IH-DH can achieve higher ATF than other hybrid on-chip memory architectures without any pure SPM, and its performance is close to or even better than that of the baseline IC-DC architecture. Therefore, we believe that among the 7 proposed hybrid on-chip memory architectures, the IH-DH architecture is the best design option to balance performance and time predictability.

5.6 RELATED WORK

Most prior work studied caches and SPMs separately. To improve time predictability of caches, researchers have proposed cache partitioning [84, 85, 86] or locking [87, 88, 89] to reduce cache interferences between tasks. However, both cache partitioning and locking may prevent dynamic reuse of cache space, which can degrade performance. In contrast, some of the hybrid SPM-cache architectures

such as the IH-DH can boost performance while improving time predictability.

Previous studies on SPM mainly treated it as an alternative to the cache memory for achieving time predictability, or energy efficiency. A number of SPM allocation algorithms have been proposed to improve either the average-case performance [78, 79, 80] or WCET [81, 82, 83]. However, since SPM is controlled by the software, a pure SPM generally is less adaptable to runtime program behavior and often leads to lower performance for general-purpose programs.

Several researchers have also explored hybrid models consisting of both cache memory and SPM, but not for time predictability. Panda et al. [78] investigated partitioning scalar and array variables into SPM and data cache to minimize the execution time for embedded applications. Verma et al. [90] studied an instruction cache behavior based SPM allocation technique to reduce the energy consumption. Recently, Cong et al. [91] proposed an adaptive hybrid cache by reconfiguring a part of the cache as software-managed SPM to improve both performance and energy efficiency. Kang et al. [92] introduced a synergetic memory allocation method to exploit SPM to reduce data cache pollution.

Comparing to all these studies that basically use a SPM to boost the performance and/or energy efficiency of an instruction or data cache, the hybrid SPM-cache architectures proposed in this paper treat both SPM and cache equally, though for different functions. More specifically, the hybrid architecture relies on the SPM to ensure a basic level of time predictability, while using caches to improve the average-case performance by exploiting the access locality for

instructions and data that cannot be stored into the SPM. Also, in this work, we have systematically and comparatively evaluated all the seven different hybrid on-chip memory architectures that can provide different tradeoffs between time predictability and performance. In addition, we believe some of the prior SPM allocation algorithms [78, 90, 91, 92] to assist the instruction or data cache are complementary to this work in terms of performance enhancement, which may be used or adapted to further improve the performance of the hybrid SPM-cache architectures provided they do not compromise the time predictability.

5.7 CONCLUSION

While cache memories are usually effective at improving the average-case performance, they are harmful to time predictability. In contrast, SPMs are time-predictable, but generally have inferior performance. To balance performance and time predictability, this chapter proposes 7 hybrid on-chip memory architectures by combining caches and SPMs to store instructions and/or data. These 7 hybrid on-chip memory architectures can provide a variety of performance and time predictability for a wide range of benchmarks. Specifically, we find that IS-DH is an attractive architecture to achieve very high time predictability while attaining performance much better than the IS-DS architecture that is purely based on SPMs. Overall, we believe IH-DH is the best hybrid on-chip memory architecture that can achieve both good time predictability and high performance. Actually, we observe that IH-DH can outperform the pure cache based architecture IC-DC for most benchmarks, revealing that improving time predictability and

performance does not have to always conflict with each other.

CHAPTER 6

CONCLUSION REMARKS

This dissertation proposes several techniques that are motivated by the unique challenges of WCET optimizations of the real-time applications:

- How can instruction prefetching on caches further improve the WCET of the real-time applications?
- How can we reduce the inter-core interferences on the shared caches in multicore processors to improve the WCET of the real-time applications?
- How can we design a time-predictable processor by a quantitative metric to reduce the complexity of the WCET analysis?
- How can we design the on-chip memories of processors to achieve both high performance and good time predictability?

Chapter 2 proposes a loop-based instruction prefetching scheme to enhance the performance for real-time applications. it can mitigate cache pollution by not prefetching instructions after the loop branches and can enhance performance by prefetching the right instructions during the loop execution. Our experimental results indicate that the loop-directed prefetching can achieve both better average-case and worst-case performance than the Next-N-Line prefetching, and thus is preferable for real-time applications.

Chapter 3 proposes three different multicore-aware code positioning approaches to either maximally reduce the longest WCET or to ensure fairness of WCET enhancement among all co-running applications by reducing the inter-core interferences on shared L2 cache. Our evaluation indicates that the WCO scheme can efficiently reduce the worst-case execution time for a single thread with the worst WCET, and the AFO and PFO schemes can reduce the WCETs of co-running threads by approximately the same amount or percentage respectively. Also, the evaluation shows that the multicore-aware code positioning approaches are generally more effective than simply separating the L2 cache by half to reduce the WCET.

Chapter 4 first presents the concept of architectural time predictability to separate the timing uncertainty concern of hardware design from software. Then we propose a new metric named architectural time-predictability factor to quantitatively evaluate architectural time predictability. The availability of such a metric allows computer architects to quantitatively evaluate the impact of different architectural/microarchitectural techniques on time predictability of processors, in addition to other important design objectives such as performance and energy dissipation, thus enabling them to make intelligent tradeoffs among time predictability, performance and energy consumption, which often conflict with each other. Our evaluation on a VLIW processor demonstrates that the proposed metric can effectively assess architectural time predictability of the processor, as well as architectural time predictability of various architectural and

microarchitectural components.

To balance performance and time predictability, Chapter 5 proposes 7 hybrid on-chip memory architectures by combining caches and SPMs to store instructions and/or data. Our experimental results demonstrate that IH-DH is the best hybrid on-chip memory architecture that can achieve both good time predictability and high performance. Furthermore, IH-DH can outperform the pure cache based architecture IC-DC for most benchmarks, revealing that it is possible to improve both time predictability and performance all together.

6.1 FUTURE WORK

Our future work of WCET optimizations lies in two aspects: In terms of software optimizations, we would like to investigate the interactions between inter-thread code positioning and intra-thread code positioning to further improve the worst-case performance and to possibly combine them for achieving the optimal results; In terms of architectural support, we would like to explore different SPM allocation algorithms for various hybrid SPM-cache architectures. Additionally, we plan to investigate the use of hybrid on-chip memory architectures in a multicore platform to balance time predictability and performance for multi-threaded and multi-programmed workloads.

REFERENCES

[1] P. Puschner and A. Burns. "Guest Editorial: A Review of Worst-Case Execution-Time Analysis," Real-Time Systems, 18(2/3):115127, May 2000.

[2] MPC500 32-bit MCU Family. Motorola/Freescale, Revised July 2002. http://- www.freescale.com/files/microcontrollers/doc/fact sheet/MPC500FACT.pdf.

[3] David Brash. The ARM architecture Version 6 (ARMv6). ARM Ltd., January 2002. White Paper.

[4] C. Berg, J. Engblom, and R. Wilhelm, Requirements for and Design of a Processor with Predictable Timing, Proc. Dagstuhl Perspectives Workshop Design of Systems with Predictable Behavior, 2004.

[5] C. Rochange and P. Sainrat, Difficulties in Computing the WCET for Processors with Speculative Execution, In Proceedings of International Workshop Worst-Case Execution Time Analysis (WCET), 2002.

[6] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, Timing Predictability of Cache Replacement Policies, Real Time Systems, vol. 37, no. 2, pp. 99-122, 2007.

[7] R. Arnold, F. Muller, D. Whalley, and M. Harmon, Bounding Worst-Case Instruction Cache Performance, In Proceedings of 15th IEEE Real-Time Systems Symposium, 1994.

[8] C.A. Healy, D.B. Whalley, and M.G. Harmon, Integrating the Timing

Analysis of Pipelining and Instruction Caching, In Proceedings of 16th IEEE Real-Time Systems Symposium, 1995.

[9] A.J. Smith, Sequential Program Prefetching in Memory Hierarchies, Computer, vol. 11, no. 12, pp. 7-21, Dec. 1978.

[10] A. Smith, Cache Memories, ACM Computing Surveys, vol. 14, no. 3, pp. 473-530, Sept. 1982.

[11] J. Smith and W.C. Hsu, Prefetching in Supercomputer Instruction Caches, In Proceedings of Supercomputing, 1992.

[12] J. Pierce and T. Mudge, Wrong-Path Instruction Prefetching, In Proceedings of 29th International Symposium of Microarchitecture (MICRO), Dec. 1996.

[13] D. Joseph and D. Grunwald, Prefetching Using Markov Predictors, In Proceedings of 24th International Symposium of Computer Architecture (ISCA), June 1997.

[14] C. Luk and T.C. Mowry, Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors, In Proceedings of 31st International Symposium of Microarchitecture (MICRO), 1998.

[15] C. Xia and J. Torrellas, Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses, In Proceedings of 23rd International Symposium of Computer Architecture (ISCA), 1996.

[16] G. Reinman, B. Calder, and T. Austin, Fetch Directed Instruction Prefetching, In Proceedings of 32nd International Symposium of

Microarchitecture (MICRO), Nov. 1999.

[17] V. Srinivasan, E.S. Davidson, G.S. Tyson, M.J. Charney, and T.R. Puzak, Branch History Guided Instruction Prefetching, In Proceedings of 7th International Conference of High Performance Computer Architecture (HPCA), Jan. 2001.

[18] P. Chow, P. Hammarlund, T. Aamodt, P. Marcuello, and H. Wang, Hardware Support for Prescient Instruction Prefetch, In Proceedings of 10th International Conference of High Performance Computer Architecture (HPCA), 2004.

[19] J. Yan and W. Zhang, WCET Analysis of Instruction Caches with Prefetching, In Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2007.

[20] S.S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997.

[21] H. Kim, Region-Based Register Allocation for EPIC Architecture, PhD thesis, New York Univ., 2001.

[22] http://archi.snu.ac.kr/realtime/benchmark/, 2010.

[23] V. Kathail, M. Schlansker, and B.R. Rau, HPL-PD Architecture Specification: Version 1.1, HPL technical report, 2000.

[24] Trimaran homepage, http://www.trimaran.org, 2010.

[25] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems, In

Proceedings of 30th International Symposium of Microarchitecture (MICRO), 1997.

[26] R. Wilhelm et al., "The Worst-case execution time problem - overview of methods and survey of tools," ACM Transactions on Embedded Computing Systems, January 2007.

[27] C. Ferdinand et al., "Precise WCET determination for a real-life processor," In Proceedings of the 1st International Workshop on Embedded Software (EMSODT 2001), Oct 2001.

[28] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley and M. G. Harmon, "Bounding pipeline and instruction cache performance," IEEE Transactions on Computers, 48(1), January, 1999.

[29] R. White, F. Muller, C. Healy, D. Whalley, and M. Harmon, "Timing analysis for data caches and set-associative caches," In Proceedings of 3rd IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 1997.

[30] I. Wenzel, B. Rieder, R. Kirner and P. Puschner, "Automatic timing model generation by CFG partitioning and model checking," In Proceddings of Design Automation and Test in Europe(DATE), March 2005.

[31] W. Zhao, D. Whalley, C. Healy and F. Mueller, "WCET code positioning," In Proceedings of 25th IEEE International Real-Time Systems Symposium (RTSS), 2004.

[32] P. Lokuciejewski, H. Falk and P. Marwedel, "WCET-driven cache-based

procedure positioning optimizations," In Proceedings of 20th Euromicro Conference on Real-Time Systems (ECRTS), 2008.

[33] C. Ferdinand and R. Wilhelm, "Fast and effiient cache behavior prediction for real-time systems," Real-Time Systems, Issue 17, 1999.

[34] J. Calandrino, D. Baumberger, T. Li, S. Hahn and J. Anderson, "Soft real-time scheduling on performance asymmetric multi-core platforms," In Proceedings of 13th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2007.

[35] J. H. Anderson, J. M. Calandrino, and U. Devi, "Real-time scheduling on multi-core platforms," In Proceedings of 12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2006.

[36] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," In Proceedings of 20th IEEE International Real-Time Systems Symposium (RTSS), 2004.

[37] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction cache," In Proceedings of 14th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2008.

[38] S. McFarling, "Program optimization for instruction caches," In Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1989.

[39] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," In Proceedings of International

Symposium on Computer Architecture, 1989.

[40] K. Pettis and R. Hansen, "Profile guided code positioning," In Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation, June, 1990.

[41] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," In Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1994.

[42] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith, "Near-optimal intraprocedural branch alignment," In Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation, June, 1997.

[43] V. Kathail, M. S. Schlansker and B. R. Rau, "HPL-PD architecture specification: version 1.1," in HPL Technical Report, 2000.

[44] S. Mohan et al., "ParaScale: exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling," In Proceedings of 21st IEEE International Real-Time Systems Symposium (RTSS), 2005.

[45] L. Gwennap, "Digital 21264 sets new standard," in Microprocessor Report, October, 1996.

[46] Homepage of Chronos, http://www.comp.nus.edu.sg/~rpembed/chronos/.

[47] Homepage of SimpleScalar, http://www.simplescalar.com.

[48] Homepage of CPLEX, http://www.ilog.com/products/cplex/.

[49] Mälardalen WCET research group, "Mälardalen wcet benchmark suite,"

http://www.mrtc.mdh.se/projects/wcet.

[50] J. Calandrino, J. Anderson, and D. Baumberger, “A hybrid real-time scheduling approach for large-scale multi-core platforms,” In Proceedings of 19th Euromicro Conference on Real-Time Systems (ECRTS), July, 2007.

[51] D. B. Kirk, “Process dependent static cache partitioning for real-time systems,” In Proceedings of 4th IEEE International Real-Time Systems Symposium (RTSS), 1988.

[52] Anonymous.

[53] D. B. Kirk, “SMART (strategic memory allocation for real-time) cache design. In Proceedings of 5th IEEE International Real-Time Systems Symposium (RTSS), 1989.

[54] D. B. Kirk, ”SMART (strategic memory allocation for real-time) cache design using the MIPS R3000,“ In Proceedings of 6th IEEE International Real-Time Systems Symposium (RTSS), 1990.

[55] A. Wolfe, ”Software-based cache partitioning for real-time applications,“ In Proceedings of 3rd International Workshop on Responsive Computer Systems, 1993.

[56] F. Mueller, ”Compiler support for software-based cache partitioning,“ In Proceedings of ACM SIGPLAN Workshop on Language, Compilers and Tools for Real-Time Systems, 1995.

[57] T. Tian and C. Shih, ”Software techniques for shared-cache multi-core systems,“ Intel Software Network, 2007.

[58] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," In Proceedings of 32nd ACM/IEEE Design Automation Conference, June 1995.

[59] Y. S. Li and S. Malik, "Cache modeling and path analysis for real-time software," In Proceedings of 12th IEEE International Real-Time Systems Symposium (RTSS), 1996.

[60] K. Kim, D. Kim and C. Park, "Real-time scheduling in heterogeneous dual-core architecture," In Proceedings of 12th International Conference on Parallel and Distributed Systems, 2006.

[61] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, Scratchpad memory: design alternative for cache on-chip memory in embedded systems, In Proceedings of the tenth international symposium on Hardware/software codesign, ser. CODES 02, 2002.

[62] M. Delvai, W. Huber, P. Puschner, and A. Steininger, Processor support for temporal predictability - the spear design example, In Proceeding of 15th Euromicro Conference on Real-Time Systems, July 2003.

[63] S. A. Edwards and E. A. Lee, The case for the precision timed (pret) machine, In Proceedings of 44th Design Automation Conference, DAC 07. , 2007.

[64] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, The Malardalen WCET benchmarks past, present and future, B. Lisper, Ed. Brussels, Belgium: OCG, July 2010.

[65] M. Kandemir and A. Choudhary, Compiler-directed scratch pad memory hierarchy design and management, In Proceedings of 39th Design Automation Conference, 2002.

[66] C. Lee, M. Potkonjak, and W. Mangione-Smith, Mediabench: a tool for evaluating and synthesizing multimedia and communications systems, In Proceedings of 30th International Symposium of Microarchitecture (MICRO), 1997.

[67] S. McFarling, Combining branch predictors, Western Research Laboratory, Tech. Rep., 1993.

[68] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero, Hardware support for wcet analysis of hard real-time multicore systems, In Proceedings of 36th International Symposium of Computer Architecture (ISCA), 2009.

[69] M. Schoeberl, Time-predictable computer architecture, EURASIP Journal of Embedded System, vol. 2009, pp. 2:12:17, January 2009

[70] L. Thiele and R. Wilhelm, Design for time-predictability, in Perspectives Workshop: Design of Systems with Predictable Behaviour, ser. Dagstuhl Seminar Proceedings, L. Thiele and R. Wilhelm, Eds., no. 03471. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[71] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, The worst-case execution-time

problem - overview of methods and survey of tools, ACM Transaction on Embedded Computing System, vol. 7, pp. 36:136:53, May 2008.

[72] N. Yamasaki, I. Magaki, and T. Itou, Prioritized smt architecture with ipc control method for real-time processing, In Proceedings of 13rd IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2007.

[73] D. Grund, Towards a formal definition of timing predictability, in Workshop on Reconciling Performance with Predictability, Grenoble, France, 2009.

[74] J. R. Daniel Grund and R. Wilhelm, A template for predictability definitions with supporting evidence, in Bringing Theory to Practice: Predictability and Performance in Embedded Systems, 2011.

[75] R. Kirner and P. Puschner, Time-predictable computing, in 8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems, Waidhofen, Austria, 2010.

[76] J. Lee et al. "Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors," In Proceedings of 12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2006.

[77] Y. Ding et al. "Architectural Time-predictability Factor (ATF): A Metric to Evaluate Time Predictability of Processors," Technical Report, Department of Electrical and Computer Engineering, Virginia Commonwealth University, April 2012.

[78] P. Panda, N. Dutt and A. Nicolau. "Efficient utilization of scratch-pad memory in embedded processor applications," In Proceedings of Europe Design and Test Conference, March 1997.

[79] S. Steinke et al. "Assigning program and data objects to scratchpad for energy reduction," In Proceedings of Europe Design and Test Conference, 2002.

[80] M. Kandemir, I. Kadayif, A. Choudhary and J. Ramanujam. "Compiler-directed scratch pad memory optimization for embedded multiprocessors," IEEE Transactions on VLSI Systems, Vol. 12, No. 3, March 2004.

[81] J. F. Deverge and I. Puaut. "WCET-directed dynamic scratchpad memory allocation of data," In Proceedings of 19th Euromicro Conference on Real-Time Systems (ECRTS), July, 2007.

[82] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. "WCET centric data allocation to scratchpad memory," In Proceedings of 21th IEEE International Real-Time Systems Symposium (RTSS), 2005.

[83] J. Whitham and N. Audsley. "Studying the applicability of the scratchpad memory management unit," In Proceedings of 16th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2010.

[84] D. Kirk. "SMART (strategic memory allocation for realtime) cache design," In Proceedings of 5th IEEE International Real-Time Systems Symposium (RTSS), 1989.

[85] D. Kirk and J. Strosnider. "SMART (strategic memory allocation for real-time) cache design using the mips r3000," In Proceedings of 6th IEEE International Real-Time Systems Symposium (RTSS), 1990..

[86] F.Mueller. Compiler support for software-based cache partitioning. SIGPLAN Notice, Vol. 30, Nov. 1995.

[87] I. Puaut and D. Decotigny. "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," In Proceedings of 18th IEEE International Real-Time Systems Symposium (RTSS), 2002.

[88] X. Vera, B. Lisper and J. Xue. "Data cache locking for higher program predictability," ACM SIGMETRICS, 2003.

[89] V. Suhendra and T. Mitra. "Exploring locking & partitioning for predictable shared caches on multi-cores. In Proceedings of 45th Design Automatic Conference, 2008.

[90] M. Verma, L. Wehmeyer, and P. Marwedel. "Cache-aware scratchpad allocation algorithm," In Proceedings of Deesign, Automation and Test in Europe Conference, 2004.

[91] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou. "An Energy-efficient adaptive hybrid cache," In Proceedings of International Symposium on Low Power Electronics and Design, 2011.

[92] S. Kang and A. Dean. "Leveraging both data cache and scratchpad memory through synergetic data allocation," In Proceedings of 18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2012.

[93] Y. S. Li and S. Malik. "Performance analysis of embedded software using implicit path enumeration," IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems, Vol. 16, Issue 12, 1997.

VITA

Graduate School
Virginia Commonwealth University

Yiqiang Ding Date of Birth: August 11, 1980

2656 Three Willows Ct, Henrico 23294

dingy4@vcu.edu

Master of Science, Department of Computer Science and Technology, Beijing University of Posts and Telecommunications, China ,April 2005
Bachelor of Science, Department of Computer Science and Technology, Beijing University of Posts and Telecommunications, China ,July 2002

Dissertation Title:
WCET Optimizations and Architectural Support for Hard Real-Time Systems

Major Professor: Dr. Wei Zhang

Publications:

JOURNAL PUBLICATIONS:

- Yiqiang Ding, Wei Zhang: Architectural Time-predictability Factor (ATF): A Metric to Evaluate Time Predictability of Processors, Accepted by ACM SIGBED Review, 2012
- Yiqiang Ding, Wei Zhang: Multicore-Aware Code Co-Positioning to Reduce WCET on Dual-Core Processors with Shared Instruction Caches, Journal of Computing Science and Engineering, Vol. 6, No. 1, pp.12-25, March, 2012
- Yiqiang Ding, Wei Zhang: Loop-Based Instruction Prefetching to Reduce the Worst-Case Execution Time, IEEE Transactions on Computers, Vol. 59, No. 6, June 2010
- Yiqiang Ding, Wei Zhang: Optimizing Instruction Prefetching to Improve Worst-Case Performance for Real-time Applications, Journal of Computing Science and Engineering, Vol. 3, No. 1, March 2009

CONFERENCE PUBLICATIONS:

- Yiqiang Ding, Wei Zhang: Static Analysis of Worst-Case Inter-Core Communication Latency in CMPs with 2D-Mesh NoC, WiP Session of LCTES 2012
- Yiqiang Ding, Wei Zhang: Multicore-Aware Code Positioning to Improve Worst-Case Performance, ISROC 2011
- Yiqiang Ding, Wei Zhang: WCET-Oriented Hybrid Code Positioning On Multi-Core Processors, ACM INTERACT-14 Workshop, 2010
- Yiqiang Ding, Wei Zhang: Improving the Static Real-time Scheduling on Multicore Processors by Reducing Worst-case Inter-thread Cache Interferences, ACM Southeast Regional Conference 2010
- Lan Wu, Yiqiang Ding and Wei Zhang: Comparatively Evaluation of Separated and Partitioned Cache Architectures for Real-time Multicore Computing, WiP Session of RTAS 2010
- Yiqiang Ding and Wei Zhang: WCET-Oriented Code Co-Positioning on Multicore Processors with Shared Instruction Caches, WiP Session of RTAS 2010