Time-predictable Execution of Embedded Software on Multi-core Platforms

SUDIPTA CHATTOPADHYAY
M.ENG(HONS), INDIAN INSTITUTE OF SCIENCE, BANGALORE

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2012

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

SUDIPTA CHATTOPADHYAY

14th September, 2012

Acknowledgements

First and foremost, I want to thank my advisor Prof. Abhik Roychoudhury for his generous guidance throughout my graduate studies. If this dissertation has taken any shape, it is due to his continuous and timely feedback on my research. I have thoroughly enjoyed all the technical discussions we have made in these four years. The quality of his guidance was way beyond my expectations. His long-term vision on research has entirely changed my viewpoint to evaluate certain aspects of research. If I ever get an opportunity to guide students, I wish that I can acquire some of his qualities to share.

I sincerely thank Prof. Khoo Siau Cheng for his teaching in the Program Analysis course. Before starting my graduate studies, I never took any course on program analysis. This dissertation revolves around designing several program analysis techniques. I truly believe that the course has built some solid foundation for me. Such a foundation on the basics of several program analysis methodologies has helped me throughout the work carried out in this dissertation.

My sincere thanks to Prof. Tulika Mitra for her research collaborations. I thank Prof. Wong Weng Fai and Prof. Liang Zhenkai for taking their time to be in my dissertation panel. I sincerely thank them for their feedback on this dissertation and helping me find some exciting directions to work in future. I thank Prof. Wang Yi from Uppsala University for his time to read my dissertation and his generous feedback.

I had a great opportunity to work with my fellow researchers in TU Dortmund, Germany. I am extremely grateful to Timon Kelter, Prof. Heiko Falk and Prof. Peter Marwedel for their excellent research collaborations. I wish to continue such collaborations for many more years in future. Finally, my sincere thanks to Lee Kee Chong for being an excellent partner in research collaborations.

It is the time to acknowledge my friends. My special thanks go to Hoang Nguyen, Shuang Liu and Sucheendra Kumar (Suchee), who are my great friends in daily life. I have bugged them with my meaningless talking and countless arguments. However, they have always been so nice

to listen to me carefully and with a lot of patience. I take this opportunity to wish them best for their future endeavor. Besides, I really appreciate the support from my friends inside and outside the lab, including Dawei, Chundong, Lee Kee, Abhijeet, Marcele, Huping, Lavanya, Pooja, Tushar, Sun Tao, Richard, Ju Lei, Eric Liang, Thuan, Chen Liang, Mihai, Malai, Bodhi, Manoranjan, Manjunath, Sriganesh and Padmanabha. I also thank my long-time friends outside NUS, specifically, Deepak Vankadaru, Raveendra Holla, Satyajit, Sayan, Debashis, Subhadeep, Arnab, Abhinav, Sarasij and Abhijit.

I thank Srivatsan Raghavan, who was my mentor and a great friend in Synopsys. Even though he was my mentor in Synopsys, he had always inspired me to pursue higher studies. Besides, I thank my friends Gaurav and Vinod with whom I had worked very closely. I wish all of them very best for their future career.

I thank School of Computing for supporting me in my conference trips. I thank all the staffs in Dean's office, specifically, Ms. Loo Line Fong, Ms. Agnes Ang and Mr. Mark Christopher for helping me in several administrative matters.

My deepest gratitude to my parents and my family. They have been always supportive throughout my graduate studies.

In this last paragraph, I shall follow the usual standard of dedicating this dissertation to one of the most important persons in my life. Throughout my academic career, my true inspiration has been my uncle, who is also my first teacher (Jagabandhu aka "Gajai"). He is the one with whom I first stepped into the academic world, even before entering any primary school. He is my first mentor. Besides, his relentless struggle in life has given me enough mental strength to survive the difficult times in my graduate studies. Over the past two decades I have seen his struggles and it has helped me learn only one thing – the importance of being indifferent towards sorrow and happiness. Probably this is the last time I shall get an open opportunity to express my deepest gratitude towards him and I do this by dedicating this dissertation to him.

Contents

Abstract

Hard real-time systems require absolute guarantees in their execution time. Worst case execution time (WCET) analysis has therefore become a very important problem to address. In recent years, multi-core processors have become widely popular due to their high performance and relatively low power consumption. With the advent of multi-core architectures, WCET prediction has become an increasingly difficult problem. The key to this problem lies in the precise and scalable modeling of shared resources, such as shared cache and shared bus. In this dissertation, we study the modeling of shared cache and shared bus for statically predicting the WCET of an application running on multi-core platform. We show that the timing predictability in multi-core can be achieved both by static analysis and compiler optimization.

We first show that the timing unpredictability due to resource sharing may also appear in single core. A meaningful example of such resource sharing in single core appears in the form of unified cache, which contains both the instruction and data memory blocks. We propose the modeling of two primary shared resources in multi-cores, namely the shared cache and the shared bus, for WCET analysis. We show that the shared cache and the shared bus have non-trivial timing interactions with pipeline and branch prediction. We propose a sound WCET analysis framework which not only models both the shared cache and shared bus, but also models the complex timing interactions of shared cache and shared bus with other basic micro-architectural components (*e.g.* pipeline, branch predictor). Our experimental results show that we can provide reasonably accurate WCET prediction and we can point the different sources of WCET overestimation. Subsequently, we show the challenges in modeling the shared cache in the presence of preemptive scheduling. We extend our WCET analysis framework with a provably correct shared cache modeling in the presence of preemptive scheduling. Apart from resource sharing, another major source of timing unpredictability in multi-core may appear due to the coherency of shared data items. In this dissertation, we have also presented a WCET analysis framework in the presence of cache coherence.

Finally, we show that the timing unpredictability in multi-core can be reduced by compiler optimization. We have studied the scratchpad allocation problem in multi-processors. We have shown that the presence of shared bus may greatly affect the scratchpad allocation decision and we have proposed a scratchpad allocation algorithm to reduce the bus traffic in multi processor system on chip (MPSoC). Our experimental results have shown that we can significantly reduce the WCET of an application compared to a scratchpad allocation algorithm which ignores shared bus delay.

In summary, this dissertation explores several technical challenges and their possible solutions for hard real-time computing in multi-cores. We believe that the methodologies and frameworks proposed in this dissertation will give valuable insights into the impact of multi-core architectures for hard real-time computing.

Related Publications

S. Chattopadhyay and A. Roychoudhury. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *IEEE Real-time System Symposium* (***RTSS***), 2009.

S. Chattopadhyay, A. Roychoudhury and T. Mitra. Modeling shared Cache and Bus in Multi-core Platforms for Timing Analysis. In *International Workshop on Software and Compilers for Embedded Systems* (***SCOPES***), 2010.

S. Chattopadhyay and A. Roychoudhury. Static Bus Schedule aware Scratchpad Allocation in Multiprocessors. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems* (***LCTES***), 2011.

S. Chattopadhyay and A. Roychoudhury. Scalable and Precise Refinement of Cache Timing Analysis via Model Checking In *IEEE Real-time System Symposium* (***RTSS***), 2011.

S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms In *IEEE Real-time and Embedded Technology and Applications Symposium* (***RTAS***) 2012.

List of Tables

List of Figures

Chapter 1

Introduction

Many of our daily life functionalities are controlled by embedded systems, such as MP3 players, washing machines, automobiles and so on. An embedded system runs a specific application for its entire lifetime. Therefore, the validation of an embedded system requires the validation of the specific application running on the system. In a way, therefore, validating an embedded system is apparently easier than validating a general purpose system, as the validation engineer has to focus upon exactly one application. On the other hand, most of the embedded software are required to satisfy some extra-functional properties, such as timing. Such time-constrained systems are not only expected to generate a *correct* output, but they are also expected to generate the correct output within *specified time bound*. These systems are also widely known as *real-time embedded systems*.

1.1 Real-time embedded systems

Real-time embedded systems can broadly be classified into two categories: *hard real-time systems* and *soft real-time systems*. Hard real-time systems need critical timing guarantees. Violation of such timing constraints for hard real-time systems may generate catastrophic effects. A typical example of a hard real-time system could be an *anti-lock braking system* (ABS) from automotive domain. An ABS is a safety system which allows the wheels of a vehicle to interact with the ground while braking and thereby avoids the vehicle to slide on the ground surface. Note that the response time of an ABS system is crucial and it needs hard real-time guarantees. The violation of such hard real-time guarantee may lead to serious consequences, such as a car accident. Soft real-time systems, on the other hand, can tolerate a certain number of violation

in timing constraints. A typical example of a soft real-time system could be a video streaming device. In video streaming, a certain amount of delay could be tolerable. As long as the inter-arrival time between two video frames does not disturb the human viewing perception, we can say that the quality of the respective video streaming device is acceptable.

1.1.1 Analysis of hard real-time systems

Since the timing constraints are critical for hard real-time systems, the timing behavior of such systems must be known at the design time. As an example, for the ABS safety system, we must know the upper bound on the brake controller's response time. In this way, the driver of the vehicle can determine a suitable position for braking and she can avoid any serious accident. The upper bound on the application response time is widely known in literature as *worst case response time* (WCRT). The analysis of worst case response time can be classified into two different levels as follows:

Worst case execution time analysis Worst case execution time (WCET) of a program gives an upper bound on the execution time of the program for all possible inputs [3]. Knowing worst case execution time is of prime importance for the WCRT analysis. Since exhaustive enumeration of all possible inputs is often infeasible, WCET is in general determined through a static program analysis. WCET analysis considers the execution time of an isolated task and the tasks are assumed never to block or to be interrupted. Blocking or interruption is taken care by the WCRT analysis. Clearly, WCET of a task depends on the underlying hardware platform and its corresponding micro-architectural parameters (e.g., size of cache, pipeline). Therefore, WCET analysis, in general estimates the execution time of a task for a given hardware platform.

System level or schedulability analysis In schedulability analysis, overall system performance is analyzed given the results of WCET analysis for each task. The scheduling could be preemptive or non-preemptive. Each task is assigned a deadline and an application is schedulable if all tasks in the application can meet their deadline. The correctness and precision of schedulability analysis thus very much depends on the accuracy and precision of WCET analysis. If WCET values are underestimated (i.e., less than the actual worst case execution time), schedulability analysis may predict an application to be schedulable even though some task may not meet its deadline in actual execution. This kind of error is considered catastrophic in hard

real-time systems and it may lead to serious consequences. On the other hand, if WCET values are pessimistic (much greater than the actual worst case execution time of task) then the scheduler will be forced to allocate more time to those tasks than actually required and thereby leading to a very imprecise schedulability analysis.

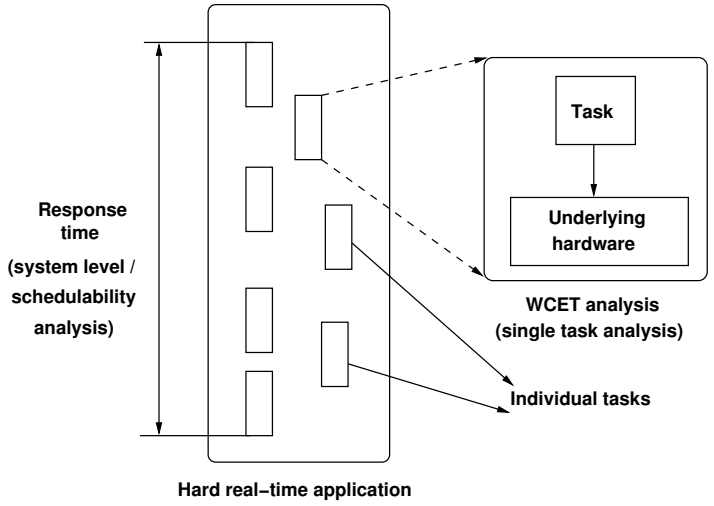


Figure 1.1: Interaction of schedulability analysis and WCET analysis

Figure 1.1 shows the dependency of schedulability analysis on WCET analysis. A real-time application typically contains multiple tasks. As shown in Figure 1.1, schedulability analysis computes the response time of the overall application. This analysis takes the WCET of each component task as input and it is usually oblivious to the low-level micro-architectural details (*e.g.* pipeline, cache). On the other hand, WCET analysis of each task is highly sensitive to the underlying micro-architecture and computes the WCET of a single task for a given hardware platform, as also shown in Figure 1.1.

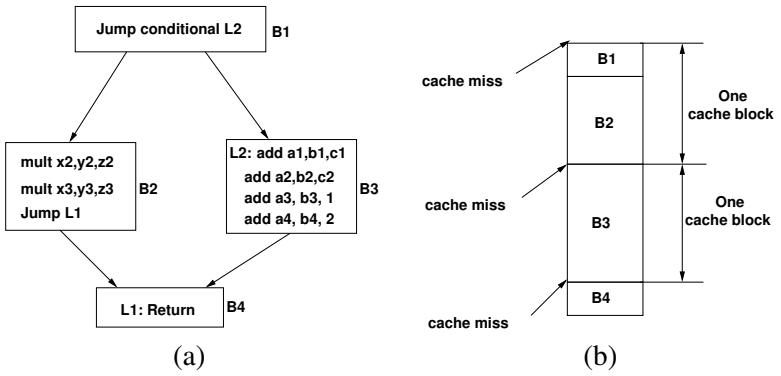


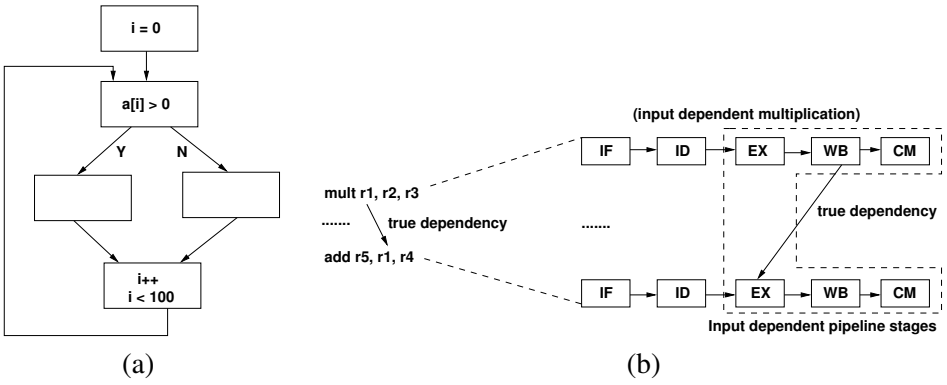
Figure 1.2: (a) A program control flow graph with two paths, (b) layout of the program code in memory, showing the instruction cache misses

The underlying hardware platform has a serious impact on the WCET of an application. Through a simple example in Figure 1.2, we shall show why the timing effects of the underlying micro-architecture cannot be ignored for a *sound* WCET analysis. Figure 1.2(a) shows the control flow graph (CFG) of a program fragment. The program fragment has exactly two paths: i) `B1-B2-B4` and ii) `B1-B3-B4`. Basic block `B2` has a set of multiplication (`mult`) instructions and basic block $B3$ has a set of addition instructions (`add`). Since multiplication is much more expensive than addition, without considering any micro-architectural effects, we can conclude that `B1-B2-B4` is the *worst case execution path*. Now consider the presence of an instruction cache and assume that the example program fragment has been loaded in memory as shown in Figure 1.2(b). If a cache block can hold four instructions, basic block `B2` will not suffer any *cache miss*. However, basic block `B3` will suffer a cache miss to load the first instruction in `B3`. As a result, the execution path `B1-B2-B4` will suffer *two* cache misses (one each at the beginning of basic block `B1` and basic block `B4`), whereas, the execution path `B1-B3-B4` will suffer *three* cache misses (one each at the beginning of each basic block). Since cache miss penalty is a magnitude higher than the processor clock cycle, `B1-B3-B4` might become the *worst case execution path*. Therefore, we conclude that the timing effects of the underlying hardware platform is of prime importance for a *sound* WCET estimate.

1.1.2 Can we use software testing to find WCET?

In general, it is infeasible to find the WCET of a task through *software testing*. Finding WCET through software testing involves executing the respective application for different inputs and recording the maximum execution time. However, it is usually expensive to enumerate all possible inputs of an application. Moreover, testing all permutations of inputs require clear domain knowledge of the application. Consider a simple program which sorts 10 given integers. Assuming a 32 bit integer, there are $2^{32*(10)}$ different permutations of inputs to the sorting program and testing the sorting program for all $2^{32*(10)}$ inputs is clearly infeasible. Similarly, consider a video processing application, such as an MPEG encoder. Finding the WCET of an MPEG encoder through software testing involves executing the MPEG encoder for a potentially unbounded number of videos. Therefore, even with sufficient knowledge of an application, it is usually infeasible to test the application for all possible inputs.

It might appear to the reader that testing all different permutations of different inputs is not necessary for finding WCET. The reader might think that it might be possible to find the WCET



(a) (b)

Figure 1.3: (a) A simple program with 2^{100} program paths, array `a` is an input, (b) a single-path program fragment where WCET cannot be obtained by executing one path

by testing all possible execution paths of a program. Such a phenomenon is commonly known as obtaining *path coverage*. For the time being, assume that WCET can be computed by obtaining path coverage. However, obtaining path coverage is also expensive, as shown by the program in Figure 1.3(a). If the array "`a`" is an input, the program has 2^{100} possible execution paths. As a result, obtaining path coverage is infeasible in practice and so as finding WCET through software testing.

Finally, we show that covering all the paths of a program may not necessarily expose the WCET of the program. Assume that we want to compute the WCET of a straight line program fragment shown in Figure 1.3(b). The program is shown at the assembly code level to demonstrate the technical issue. Figure 1.3(b) also shows the execution of the program fragment in a conventional five stage pipeline (*i.e.* `IF-ID-EX-WB-CM`). The result produced by the multiplication instruction (*e.g.* `mult`) is used by the addition (*e.g.* `add`). This dependency is shown by the solid edge between the `WB` and the `EX` pipeline stages of the two instructions. In general, a multiplication instruction has variable latencies (which depends on the operands). Therefore, if the multiplication instruction is *input dependent*, the `EX` stage of the addition instruction will finish at different time points (depending on the concrete input values), which in turn will affect the WCET of the program fragment. Such input dependent pipeline stages are highlighted in Figure 1.3(b). However, note that it was sufficient to obtain program path coverage of the program fragment by generating just *one* concrete input. In general, for a single program path, there are different execution scenarios at the micro-architectural level. One such example is shown in Figure 1.3(b). However, in the presence of advanced micro-architectural features (*e.g.* superscalar and out-of-order pipeline, speculative execution) the situation is far more complex

and a single program path may lead to a huge number of execution scenarios depending on the concrete values of inputs.

Therefore, we conclude that conventional software testing *may miss* the program input for which the actual worst case execution time of the program is reached. Consequently, a *sound* WCET (*i.e.* a true upper bound on the execution time over all possible inputs) of a program cannot be obtained solely by software testing. On the other hand, a conservative static program analysis technique can analyze the program irrespective of the program's input. Consequently, it is possible to obtain a *sound* upper bound of the actual WCET of a program through static program analysis.

1.2 Motivation and thesis overview

As we describe in the preceding, WCET analysis of a program depends on the underlying hardware platform. With the advent of multi-core architectures, multi-core processors have been adopted for mainstream computing due to the high performance and low power consumption offered by such processors. Therefore, it is reasonable to adopt the advantages of multi-core processors in the area of real-time embedded systems. Whereas multi-core processors offer several performance and power related advantages, they pose significant challenges to be adopted for hard real-time systems. Multi-core processors extensively employ shared resources (*e.g.* shared cache, shared bus). The presence of shared resources drastically increases the timing unpredictability, as the timing largely depends on the nature of conflicts generated in the shared resources. The conflicts in the shared resource, on the other hand, highly depend on the applications running on different cores. To adopt multi-core processors for hard real-time systems, it is necessary to predict the execution time of an application. In this dissertation, we have performed an in-depth study of WCET analysis of programs running on multi-core platforms. Our modeling of shared cache and shared bus can be used for statically predicting the WCET of an application running on multi-core platforms. Moreover, our WCET analysis framework can be used to pinpoint the sources of WCET overestimation and it can be used for specialized compiler optimizations to achieve time predictability. As evidenced by our scratchpad allocation optimization, such an analysis framework can help to reduce the timing unpredictability arising due to a shared bus in multi-core.

1.3 Organization of the chapters

In this dissertation, we concentrate on static WCET analysis in the presence of multi-core architectures. Multi-core architectures extensively employ shared resources (e.g., shared cache, shared bus etc) and introduction of shared resources makes WCET analysis a very challenging problem due to the unpredictable conflicts generated in the same. Among others, shared cache and shared bus are two meaningful examples of resource sharing in multi-cores (where the conflicts are generated at the level of memory references from different tasks running on different processor cores). Resource sharing can also be found in single-core architectures. Unified cache, where the conflicts are generated at the level of instruction and data memory blocks, is a key example for resource sharing in single-core architecture. We therefore introduce our work by modeling the unified cache for WCET analysis in Chapter 4 and later move on to model the shared cache and the shared bus in multi-cores in Chapter 5 and Chapter 6, respectively. In Chapter 7, we propose a unified WCET analysis framework which models the complex timing interactions between the shared resources and other basic micro-architectural components (*e.g.* pipeline, branch predictor). In Chapter 8, we extend our framework in the presence of preemptive multi-tasking systems. In Chapter 9, we show that the timing unpredictability in multi-core system may also appear due to the coherency of data items shared by multiple cores. We also present a WCET analysis framework in the presence of cache coherence in Chapter 9. Finally, in Chapter 10, we shall describe how our analysis framework can be used to reduce the conflicts in shared resources through a compiler optimization pass. More specifically, we show that the shared bus traffic can be reduced by selecting appropriate contents in software controlled scratchpad memories. In Chapter 11, we shall conclude this dissertation and discuss possible future directions.

Chapter 2

WCET Analysis Background

In this Chapter, we shall introduce a general background on *worst case execution time* (WCET) analysis. We shall discuss the different phases required for static prediction of WCET, as well as the related technical issues associated with each of these phases.

2.1 Static WCET analysis

Static WCET estimation typically involves three phases: program flow analysis (to find the infeasible program path and loop bound), micro-architectural modeling (to find the timing effects of underlying hardware) and a calculation phase to find the longest feasible program path using the results of program flow analysis and micro-architectural modeling. In the following, we shall briefly discuss each of these three analysis phases.

Program flow analysis The goal of program flow analysis is to find infeasible program paths and loop bounds. The *soundness* of WCET analysis is not affected by infeasible program paths. However, with the knowledge of infeasible paths, the static WCET analyzer can ignore certain paths during WCET computation. This in turn may lead to a tighter WCET estimation. Consider an example program and its corresponding control flow graph (CFG) shown in Figure 2.1. Without any knowledge of infeasible path, assume that the WCET analyzer computes `B2-B3-B5-B6-B8` as the worst case path inside the loop. However, careful examination reveals that the condition of basic block `B2` (*i.e.* $z == 0$) and basic block `B5` (*i.e.* $z < -2$) cannot be satisfied together for any execution. Therefore, `B2-B3-B5-B6-B8` captures an *infeasible execution* and therefore, it can be ignored during the WCET analysis. In general, if such infeasible path information can be integrated into a WCET analyzer, the analysis may lead to a more

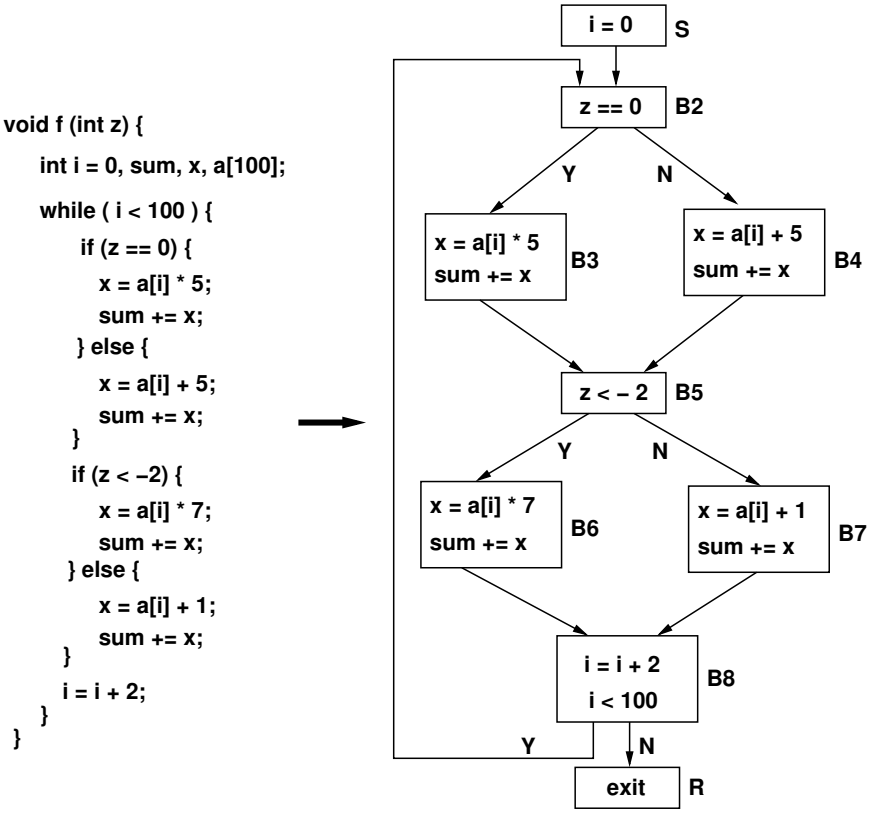


Figure 2.1: An example program and its corresponding control flow graph (CFG)

precise WCET estimate by focusing on a lesser number of possible execution paths.

Whereas the discovery of infeasible paths may only affect the precision of WCET analysis, WCET prediction is not possible without knowing the upper bound of all loop iterations in the program. In the example shown in Figure 2.1, it is not possible to predict the WCET of function `f` without knowing that the loop iterates 50 times. Therefore, discovering the upper bound on loop iteration is potentially more important for an accurate WCET prediction.

The research on flow analysis has focused on automatic discovery of infeasible paths as well as loop bounds [4; 5; 6; 7]. If the upper bound on loop iteration cannot be inferred statically, such an upper bound can be provided manually to the WCET analyzer in the form of user annotations. Similarly, certain infeasible program paths might be provided manually to the WCET analysis tool to get a tighter WCET estimation.

Micro-architectural modeling WCET of an application is highly sensitive to the underlying hardware platform. Therefore, to predict a *sound* and *precise* WCET of an application, the timing effects of the underlying hardware need to be modeled. Micro-architectural modeling analyzes the timing effects of underlying hardware components (*e.g.* pipeline, cache, branch

predictor etc) and it is the crucial part of WCET analysis process. In past two decades, an extensive amount of research effort has been put forward for micro-architectural modeling. One of the first few approaches include the use of integer linear programming (ILP) [8], but the use of ILP poses scalability issues due to the presence of a huge number of ILP constraints. However, the breakthrough in micro-architectural modeling was first proposed in [9]. The work in [9] has proposed a scalable approach of using abstract interpretation for micro-architectural modeling. Since its inception [10], abstract interpretation has been successfully applied to many application domain including functionality testing and compiler optimization. In [9], abstract interpretation was proposed to be used for WCET analysis. The basic framework proposed in [9] has later been extended by many research efforts to analyze advanced micro-architectural features, such as data cache [11], multi-level cache [12], pipeline [13], branch predictor [14], shared cache [15] and so on.

Micro-architectural modeling for modern processors is complicated due to a commonly known phenomenon called *timing anomaly* [16]. Assume a sequence of instructions containing a particular instruction I. Further assume that I have two possible latencies L_1 and L_2, which lead to a total execution time of E_1 and E_2, respectively, for the sequence of instructions. Note that I might have variable latencies due to different reasons, such as, *cache hit/miss*, variable execution cycle (*e.g.* multiplication instruction) and so on. A *timing anomaly* occurs when $L_1 < L_2$, but $E_1 > E_2$. Timing anomaly is best explained by an example. We shall use the example shown in Figure 2.2 to illustrate the problem.

Figure 2.2(a) shows a sequence of multiplication instructions and its execution in a multiple-way, superscalar processor. The fourth instruction has a dependency on the third instruction due to the computation in register `r8`. Additionally, for the sake of illustration, we assume the following:

- Multiplication has variable execution latency $1 \sim 4$ cycles. First three multiplication instructions take 4 cycles to execute and the fourth instruction takes 3 cycles to execute.
- Cache miss penalty is 6 cycles.
- There are a total of two multiplier units.

We shall consider two execution scenarios: (EX_1) the first instruction is an *instruction cache hit*, and (EX_2) the first instruction is an *instruction cache miss*.

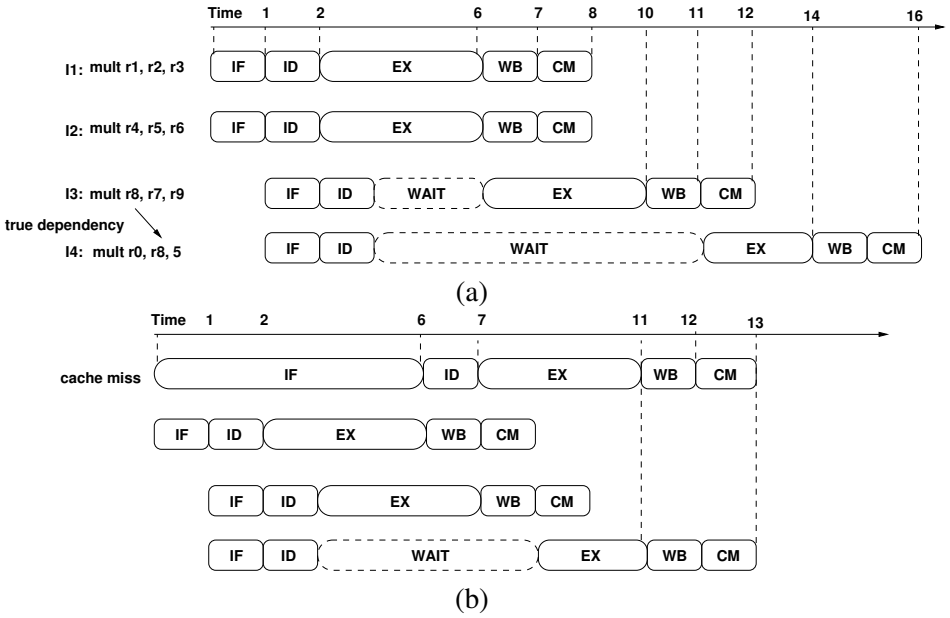


Figure 2.2: An example showing timing anomaly. (a) Execution scenario with $I1$ facing instruction cache hit, (b) execution scenario with $I1$ facing instruction cache miss

In EX_1 (shown in Figure 2.2(a)), instruction $I3$ has to wait till 6th cycle as the two multiplier units are occupied by $I1$ and $I2$. Since $I4$ depends on the result computed by $I3$, $I4$ also has to wait for $I3$ to finish execution. Eventually, the sequence of instructions $I1, I2, I3, I4$ finishes in 16 cycles.

Now consider the second execution scenario where $I1$ is an instruction cache miss (shown in Figure 2.2(b)). In this case, $I3$ can finish execution at 7th cycle using one of the free multiplier units. Subsequently, $I4$ can finish execution at 11th cycle and the sequence of instructions finishes in 13 cycles.

From the above example, we observe that a *cache hit* (which is a local worst case scenario) leads to an overall *worse* execution time compared to a *cache miss*. Such counter intuitive phenomenon appears due to the complex timing interactions between cache and pipeline. The example in Figure 2.2 also demonstrates that it is insufficient to track the local worst case of each instruction (such as a cache miss rather than a cache hit) to compute the WCET of an entire program. As a result, to compute the WCET of a program, one needs to keep track of the different micro-architectural states and their possible permutations. However, capturing all possible micro-architectural states is, in general, infeasible. Therefore, existing works use abstract micro-architectural states via abstract interpretation [9; 13] or timing interval abstraction to capture the time taken by each pipeline stage [17; 18].

Path analysis Path analysis uses the results by program flow analysis and micro-architectural modeling to find the longest feasible program path in the program. Among others, *path-based technique* and *implicit path enumeration* are mostly used for the calculation of WCET.

Path-based techniques try to find the WCET of the program by enumerating feasible program paths and then searching for the program path having longest execution time. Path-based techniques are naturally very precise and these techniques can also integrate various program flow information (computed during flow analysis) while searching for the longest path. Path-based WCET calculation has been used in [19]. However, path-based techniques suffer from scalability problem, as it enumerates a huge number of paths. The work of [20] somewhat addresses this issue by systematically removing the infeasible paths from the control flow graph.

Implicit path enumeration techniques represent program control flow as linear equations/-constraints and formulate the WCET computation problem as maximizing the objective function of an integer linear program (ILP). The solution of the ILP can be derived by any ILP solver (*e.g.* CPLEX [21]). The solution of the ILP contains a quantitative value capturing the WCET of the program and the execution count of different control flow edges. However, the solution of the ILP does not return the exact execution path which leads to the worst-case scenario. The work of [9] first comprehensively combined the abstract interpretation based micro-architectural modeling and the ILP-based path analysis for WCET computation. Moreover, most of the common forms of program flow information (such as infeasible path, loop bound) can easily be encoded as linear constraints and they can be integrated into the WCET formulation (as shown in [5; 22]). Consequently, ILP-based WCET computation has become popular in the research community. Many WCET analyzers currently employ ILP-based (such as Chronos [23], aiT [24]) calculation phase.

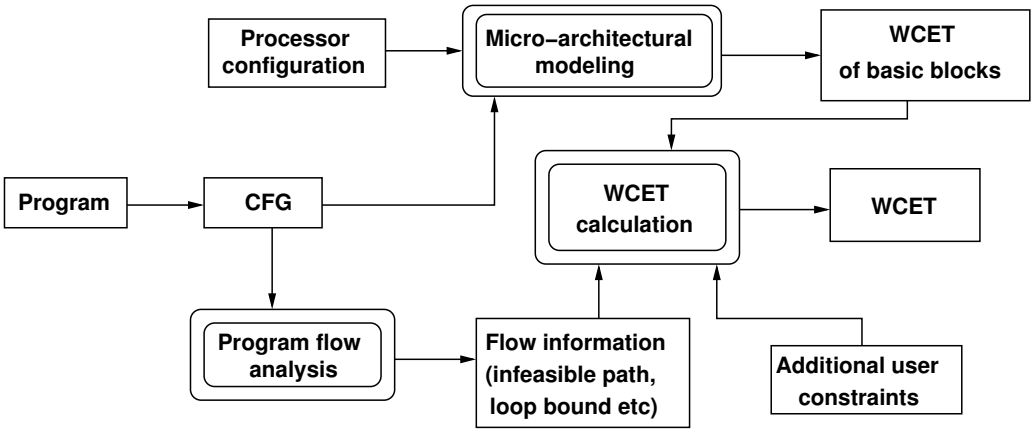


Figure 2.3: Overview of a typical WCET analysis framework

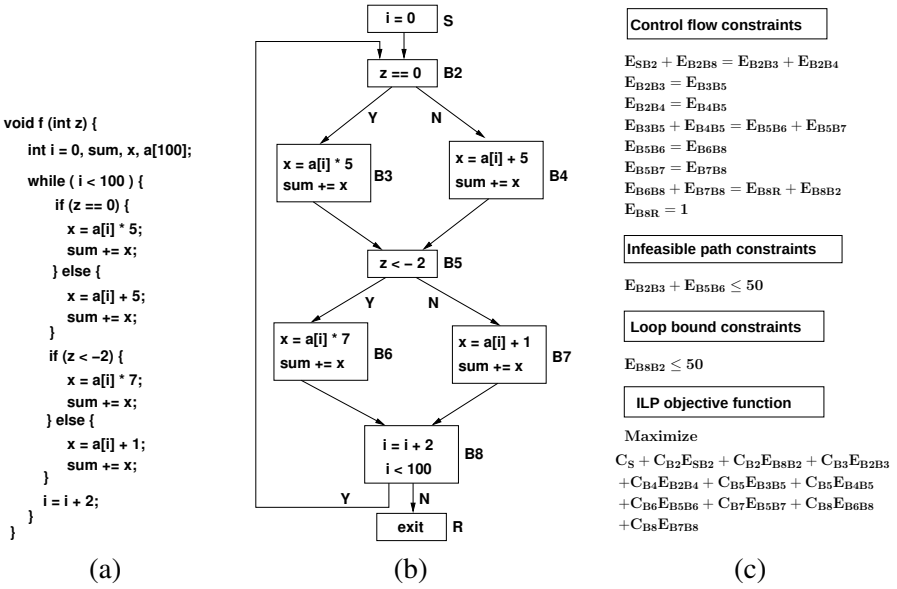


Figure 2.4: An example showing ILP-based WCET calculation

Figure 2.3 captures an overview of a typical WCET analysis process. Micro-architectural modeling usually works at the level of basic blocks and computes the WCET of each basic block. Program flow information can be derived by static analysis and some additional flow information can also be given by the user manually. WCET of each basic block and program flow information (loop bound, infeasible path) are used to compute the WCET of the entire program, as shown in Figure 2.3.

2.2 Example

In this section, we shall illustrate the WCET computation by revisiting the example shown in Figure 2.1. We shall use the *implicit path enumeration based* WCET calculation for the illustration.

The example is shown in Figure 2.4(a). Although, WCET analysis is usually carried out on the executable, for the sake of simplicity in the discussion, we shall show the process at the source code level. Control flow graph of the program is shown in Figure 2.4(b).

Let us assume, C_B denotes the WCET of basic block B derived via micro-architectural modeling. Further assume E_{B1B2} is the ILP variable which denotes number of times the edge from basic block $B1$ to basic block $B2$ is taken in the execution. Therefore, we have the following objective function in the ILP formulation:

$$\textbf{Maximize} \quad C_S + C_{B2}E_{SB2} + C_{B2}E_{B8B2} + C_{B3}E_{B2B3} + C_{B4}E_{B2B4} + C_{B5}E_{B3B5} + C_{B5}E_{B4B5} + C_{B6}E_{B5B6} + C_{B7}E_{B5B7} + C_{B8}E_{B6B8} + C_{B8}E_{B7B8} \tag{2.1}$$

Representing control flow and loop bound Only one execution path is taken at a branch. Therefore, we have a set of control flow constraints as shown in Figure 2.4(c). The program in this example contains a loop and for WCET computation, the loop bound must be known. For the example program, the upper bound on the loop iteration is 50. This loop bound can be explicitly specified by the user or it can also be derived through a complex analysis of the program (e.g., using [7]).

Representing infeasible path Certain infeasible path informations can be represented as linear constraints and therefore, they can easily be integrated into the ILP-based calculation. Note that both the basic blocks `B3` and basic block `B5` cannot be present in any feasible execution. This is due to the infeasible condition $z == 0 \wedge z < -2$. Such infeasible paths can be represented as linear constraints as shown in Figure 2.4(c).

An ILP solver (*e.g.* CPLEX) maximizes the objective function (as specified in Equation 2.1) considering all specified constraints to it (Figure 2.4(c)).

2.3 Chapter summary

In this chapter, we have briefly discussed the different steps involved in a typical WCET analysis process. As the work in this dissertation is concentrated towards multi-core architectures, our work mainly revolves around multi-core specific micro-architectural modeling, while keeping the rest of the phases of a WCET analysis framework mostly unchanged.

Chapter 3

Literature Review

In this Chapter, we present an overview of the existing research in both single and multi-core. As the existing research on shared caches are based on the cache analysis in single core, we first start with an overview of cache analysis in single core. Subsequently, we shall discuss the existing research in shared caches and shared buses in multi-core. Finally, we shall give an overview of the existing memory optimization techniques for improving execution time predictability.

3.1 Cache analysis of a single task

Most of the research in cache modeling consider a single level of instruction cache. Among others, abstract interpretation based cache analysis proposed in [9] deserves mention. For analyzing the cache behaviour of a single task, *must* and *may* cache analysis has been proposed in [9]. *Must* and *may* cache analysis categorize memory references as *all-hit* (AH) and *all-miss* (AM) respectively. The memory block corresponding to an AH categorized memory reference is always in cache when accessed. On the other hand, the memory block corresponding to an AM categorized memory reference is never in cache when accessed. *Must* analysis can be used along with virtual inline and virtual unrolling (VIVU) to significantly improve the analysis precision. In VIVU approach, each loop is unrolled once to distinguish the *cold cache misses* at first iteration of the loop. If a memory reference cannot be classified as AH or AM, it is considered *unclassified* (NC). For a *fully timing composable* architecture, an NC categorized memory reference can be considered as a *cache miss* during worst-case computation. The analysis proposed in [9] has later been extended to analyze multi-level non-inclusive instruction caches (in [12]).

Static analysis of data cache timing effects have also been studied (*e.g.*, see [11; 25]). In particular, [11] adapts the abstract interpretation approach for data cache analysis. One of the major difficulties in data cache analysis is the fact that several executions of an instruction can access different data memory addresses and it is difficult to precisely predict the range of data memory addresses accessed by a particular instruction. The work proposed in [26] addresses this concern somewhat by partial unrolling of loops. A recent approach [27] improves over the state-of-the-art data cache analysis by employing a scope based cache state computation. The work in [27] is based on the insight that a data memory reference may access different memory blocks in different iterations of a loop. For each memory block m accessed by a particular load/-store instruction and for each loop nesting depth, [27] defines a set of iteration interval (called as *temporal scope*) in which m could be accessed. A temporal scope $L \mapsto [x, y]$ of memory block m captures that m can *only* be accessed between iteration x and iteration y of loop L, but m can *never* be accessed *before iteration* x and *after iteration* y *of loop* L. Such a temporal scope based partitioning is quite useful for data cache analysis, as different memory blocks accessed by a load/store instruction may have totally *disjoint* temporal scopes and therefore, may not conflict in the cache with each other. Once the temporal scopes are computed for each data reference instruction, [27] employs a scope based *persistence* analysis. Such a persistence analysis classifies each memory block accessed by a data reference as *persistence* (PS) or *unclassified* (NC) with respect to a loop nesting depth.

In summary, all existing works on cache modeling focus on either instruction cache or data cache, but not both. Moreover, for cache hierarchies, in most real processors the second-level cache is a unified cache which contains both instruction and data — an issue not considered in existing works. In Chapter 4, we show that the timing unpredictability may arise due to the sharing in unified cache — at the level of different instruction and data memory blocks. In our work (described in Chapter 4), we build on existing works to develop a WCET analyzer which considers separate instruction and data caches in the first level and a unified cache in the second level.

3.2 Inter-task cache conflict analysis

Inter-task cache conflict analysis is required to find an upper bound on cache misses due to preemption. The bound on cache misses (or additional clock cycles) due to preemption is called

cache related preemption delay (CRPD). In last decade, there has been an extensive amount of research to bound the cache related preemption delay (CRPD) [28; 29; 30; 31; 32; 33; 34]. There are three main approaches to statically bound the value of CRPD:

- Analyzing the preempted task ([28; 33]),
- Analyzing the preempting task ([30]), and
- Analyzing both the preempted and the preempting task ([29; 34]).

The analysis of the preempted task revolves around the concept of *useful cache block* (UCB) [28]. A UCB is a block that may be *cached* before preemption and may be *used* later, resulting in a cache hit in the absence of preemption. A *data flow analysis* is applied on the preempted task to statically predict the set of UCBs at each program point. The set of UCBs poses an upper bound on the *additional cache misses* for a single preemption. Recently, [33] has improved the state-of-the-art CRPD analysis [28] by reducing the number of UCBs to consider for CRPD computation. The key idea of [33] is based on the observation that CRPD analysis is always used along with the WCET analysis. Therefore, the technique proposed by [33] considers only those cache misses which were not predicted cache miss by the WCET analysis. In this fashion, [33] may not be able to preserve the over-estimation of CRPD in isolation, however, it can guarantee the over-estimation of the sum of WCET and CRPD.

The analysis of the preempting task is based on the notion of *evicting cache block* (ECB). The set of cache blocks used by the preempting task during its execution is known as ECB. If a cache set is unused by the preempting task, it cannot evict any of the cache blocks used by the preempted task in the respective cache set. Therefore, researchers have proposed to use the set of ECBs for estimating CRPD in [30; 31].

[29] has proposed a precise CRPD analysis approach based on the combination of UCB and ECB. Therefore [29] analyzes both the preempted task (for computing the UCBs) and the preempting task (for computing the ECBs). A UCB may lead to an additional cache miss after preemption only if it might be evicted by an ECB. Such a CRPD analysis framework [29] is more precise than the CRPD analysis based on analyzing either the preempted or the preempting task in isolation. However, the analysis of [29] is based on *direct-mapped* caches. As shown in [34], set-associative caches introduce additional complications in accurately estimating the set of UCBs that can be replaced by a set of ECBs. [34] proposes a CRPD analysis framework for general, set-associative caches.

[32] shows that the precision of CRPD analysis does not only depend on the precision of UCB and ECB, but it also depends on the set of preemption points. The technique proposed by [32] is based on the following insight: if two different preemptions at p and p' may lead to a cache miss of the same memory reference in the preempted task, then we need to consider only one additional cache miss in the preempted task for the set of preemption points $\{p, p'\}$. This could be possible, only if the analyzer has the knowledge of both the preemption points p and p'. On the other hand, if we compute the CRPD for p and p' in isolation, it will lead to consider duplicate cache misses for the same memory reference in the preempted task. Computing the CRPD for *all possible set of preemption points* will lead to an exponential slow-down. Therefore, [32] proposes efficient algorithms which account multiple preemption points to improve the precision of *state-of-the-art* CRPD analysis.

In summary, there has been an extensive set of works to estimate CRPD based on UCB and ECB. Several improvements over the state-of-the-art by combining UCB and ECB (*e.g.* [29; 34]) and maintaining the knowledge of multiple preemptions (*e.g.* [32]) have also been proposed in the previous years. However, all of the previous works target a single level cache. On the contrary, we leverage the concept of both UCB and ECB in the context of cache hierarchy in multi-core (described in Chapter 8). To the best of our knowledge, *ours is the first CRPD analysis framework* which targets a cache hierarchy and provides an analysis framework to bound the value of CRPD in the presence of shared caches in multi-core.

3.3 Shared cache analysis

In multi-core systems, tasks in different cores may execute in parallel while sharing memory space in the cache hierarchy. Wei Jhang and Jun Yan [35] were first to introduce the shared cache modeling for software timing analysis. In this work they differentiate a memory block inside some loop or outside any loop. The underlying architecture has two levels of cache where the second level is shared across cores. All memory blocks, which are present in the private cache of a core, do not suffer from the conflicts introduced by the threads running in other cores. Similarly, only L2 cache hits are required to change since L2 cache misses already exploit the worst-case scenario for WCET analysis. Therefore, only L1 cache misses and L2 cache hits are analyzed further to detect possible conflicts introduced by other cores. The analysis works as follows: assume a task T accesses a memory block m and is mapped to cache set C in the L2

cache. Task T' is concurrently running with T in a different core and therefore share the L2 cache with task T. Further assume M' is the set of memory blocks accessed in T' which maps to same cache set C in L2 cache and memory block m was a cache hit in L2 cache ignoring the conflicts from other cores. The classification of memory block m is changed to a *L2 cache miss* if any of the memory blocks inside the set M' is accessed inside a loop of T'. On the other hand, the classification of memory block m is changed to a *always-except-one L2 cache hit* if M' contains a single memory block and is accessed outside any loop of T' (since the memory block inside M' can be accessed at most once).

The analysis proposed in [35] has several limitations: first, the approach does not exploit the task dependency. On the other hand, real life embedded applications generally contains multiple tasks and the dependency could be provided through a task graph or message sequence chart [36]. Two dependent tasks can never interfere in the shared resources. Similarly, two different tasks never interfere if their execution times do not overlap. This observation was first made in a recent work proposed in [15]. [15] has proposed an iterative WCRT (Worst Case Response Time) analysis framework for modeling shared cache. Conflicts in shared cache depends on task lifetimes which on other hand depends on shared cache contents of each task. To resolve this dependency, an iterative framework was proposed in [15]. Authors of [15] have also formally proved that their framework always terminates after a finite number of iterations. Secondly, [35] does not employ any additional optimizations for set associative caches. On the other hand, the shared L2 cache is normally made set associative to reduce conflicts. [15] solves this problem with LRU replacement policy. It simply checks, how much more a memory block can grow in a shared cache set in age without conflicts from other cores. Therefore, if number of memory blocks (from the conflicting tasks) mapping to the same cache set is not more than this limit, the corresponding memory block remains to be a cache hit in the shared cache even after conflicts from other cores.

Hardy et al. [37] have proposed a compile time optimization to improve the precision of [35]. The work of [37] performs a compile time analysis to find memory blocks which are used at most once in the program. Since the memory blocks are used only once, they do not heavily affect the performance of the program, however, they might evict some memory blocks from the shared cache which are used heavily by other cores. Therefore, the central idea was to restrict all such memory blocks (used only once) to go into the shared L2 cache.

Existing works on shared caches suffer from overestimating the infeasible inter-core cache

conflicts. In Chapter 5, we propose a novel approach which combines abstract interpretation and model checking for building a *scalable* as well as *precise* WCET analysis framework.

3.4 Shared bus modeling

Shared bus analysis introduces several difficulties in accurately analyzing the variable bus delay. It has been shown in [38] that a time division multiple access (TDMA) scheme would be useful for WCET analysis due to its statically predictable nature. Subsequently, the analysis of TDMA based shared bus was introduced in [39]. In [39], it has been shown that a statement inside a loop may exhibit different bus delays in different iterations. Therefore, all loop iterations are virtually unrolled for accurately computing the bus delays of a memory reference inside loop. As loop unrolling is sometimes undesirable due to its inherent computational complexity, in [40] (described in Chapter 6), we have proposed a TDMA bus analysis technique which analyzes the loop without unrolling it. Moreover, our work, as described in Chapter 6 was the *first to propose an analysis framework which models both the shared cache and shared bus*. However, [40] requires some fixed alignment cost for each loop iteration so that a particular memory reference inside some loop suffers exactly same bus delay in any iteration. The analysis proposed in [40] is fast, as it avoids loop unrolling, however imprecise due to the alignment cost added for each loop iteration. Finally, [41] proposes an efficient TDMA-based bus analysis technique which avoids full loop unrolling, but it is almost as precise as [39]. The analysis time in [41] significantly improves compared to [39]. However, none of the works ([39; 40; 41]) model the interaction of shared bus with pipeline and branch prediction. Additionally, [39] and our work in [40] assume a *timing-anomaly-free* architecture. A recent approach [42] has combined abstract interpretation and model checking for WCET analysis in multi-cores. The micro-architecture analyzed by [42] contains a private cache for each core and it has a shared bus connecting all the cores to access main memory. The framework uses abstract interpretation ([9]) for analyzing the private cache and it uses model checking to analyze the shared bus. However, [42] ignores the interaction of shared bus with pipeline and branch prediction. It is also unclear whether the proposed framework remains scalable in the presence of shared cache and other micro-architectural features (*e.g.* pipeline). In Chapter 7, we propose a unified WCET analysis framework for multi-core platforms. Such a unified WCET analysis framework models both the shared cache and shared bus. Additionally, our framework models the complex interactions with shared cache

and shared bus with pipeline and branch predictor, without appreciable loss of efficiency.

3.5 Time predictable micro-architecture and execution model

To eliminate the problem of pessimism in multi-core WCET analysis, researchers have proposed predictable multi-core architectures [43] and predictable execution models by code transformations [44]. The work in [43] proposes several micro-architectural modifications (*e.g.* shared cache partitioning among cores, TDMA round robin bus) so that the existing WCET analysis methodologies for single cores can be adopted for analyzing the hard real-time software running on such system. On the other hand, [44] proposes compiler transformations to partition the original program into several *time-predictable* intervals. Each such interval is further partitioned into *memory phase* (where memory blocks are prefetched into cache) and *execution phase* (where the task does not suffer any last level cache miss and it does not generate any traffic to the shared bus). As a result, any other bus traffic scheduled during the execution phases of all other tasks does not suffer any additional delay due to the bus contention.

3.6 Memory optimization for execution time predictability

3.6.1 Cache locking and cache partitioning

Due to the difficulty in analyzing cache behaviors, several analyses pose restrictions on parts of the program for predictable execution time behaviour. *Cache locking* and *cache partitioning* are two meaningful examples of such transformation. In cache locking, users are allowed to load the content in the cache and subsequently prevent those content to be evicted out from the cache. Cache locking is available as an ISA feature in several commercial processors (e.g., ARM 920T, PowerPC 440 Core etc). However, placement of this cache locking instruction is crucial and wrong placement (which may happen when cache locking instructions are placed manually) of lock instruction may severely degrade application performance by blocking frequently used memory blocks to be loaded into the cache. Therefore, researchers have investigated automatic placement of cache locking instruction to improve worst case performance. Among others, Puaut's work in [45] for software controlled cache deserves mention. The work of [45] divides program code with respect to different function entries and loop headers (called reload points in [45]). The portion of code between two reload points is called a program region. The approach

described in [45] statically computes the frequently accessed memory blocks along the worst case path in a region and locks the cache after loading those memory blocks before entering the same region.

When cache locking is mainly used for eliminating intra task interferences, cache partitioning techniques are mainly used for eliminating inter task interferences. This situation arises when multiple tasks, with overlapping execution time, may use the same cache. For example, in previous section, we discussed about CRPD analysis to compute the additional cache reloading delay encountered due to the preemption. One can eliminate the use of such analysis by assigning different disjoint partitions to different tasks. Cache partitioning techniques have been proposed in [46; 47] to guarantee that the most recently used memory blocks will remain in the cache despite preemption. Therefore, these techniques will be able to eliminate inter task interferences. However, cache partitioning may have the disadvantage of exploring only a limited amount of space in the cache and therefore may encounter more cache misses due to the intra task interferences. A combination of cache locking and cache partitioning for average case performance improvement has also been proposed in [48; 49] for data caches. In [49], first the cache is partitioned among different tasks. Each portion of the program that are difficult to analyze (e.g., have unpredictable data memory accesses), are locked by using a greedy heuristic. The program regions with predictable data memory accesses can be analyzed using static data cache analysis (*e.g.* [11]). Therefore, the work proposed in [49] comprehensively combines cache partitioning, cache locking and static cache analysis for better time predictability. The work in [4] has subsequently explored several combinations of cache locking and cache partitioning for shared caches used in multi-core architectures.

3.6.2 Changing layout of memory blocks

Changing the layout of memory blocks may lead to predictable execution time behaviour. Most of the processors incur a pipeline delay whenever an instruction transfers control to a target that is not the next sequential instruction. Compiler developers therefore try to place the most frequently used memory blocks in the memory sequentially to reduce average case execution time (ACET). Zhao et. al. [50] first proposed the equivalent optimization to reduce worst case execution time (WCET). In [50], most frequently accessed basic blocks along the worst case execution path (WCEP) are placed in contiguous memory location to improve WCET. Additionally, placement of basic blocks along the WCEP increases the spacial locality along the

WCEP and therefore also improves the worst-case cache behaviour. A recent paper [1] places the most frequently called functions along the WCEP contiguously to reduce cache conflict misses and thereby improving the worst case performance. In summary, all of these works have optimized the layout of code ignoring the layout of data. In our work ([51]), we show that changing only the layout of code may lead to performance degradation in presence of unified cache. Therefore, we proposed a novel algorithm in [51] which changes the layout of instruction and data simultaneously to improve WCET. We describe this optimization in Section 4.6.

3.6.3 Scratchpad memory

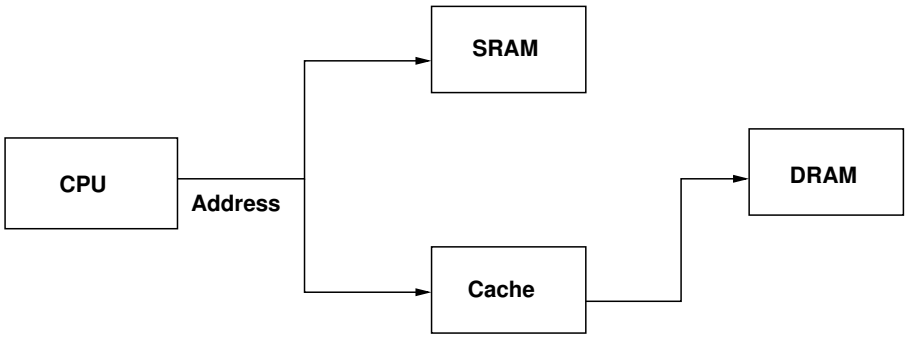


Figure 3.1: Addressing mechanism in the scratchpad memory and the cache

Scratchpad is a fast on-chip memory and it is explicitly controlled by user or managed by system software (*e.g.*, a compiler). Scratchpad memory is mapped into the address space of the processor. Whenever the address of a memory access falls within a pre-defined range, the scratchpad memory is accessed instead of caches. This addressing mechanism is demonstrated in Figure 3.1. Since scratchpad memory contents can be controlled by a compiler, each memory access to scratchpad becomes predictable. Therefore, scratchpad memory has widely been adopted for real-time embedded systems instead of caches where the memory management is entirely transparent to the user. Moreover, it is also shown in [52] that using scratchpad memory leads to a reduced area and energy consumption compared to caches. However, use of scratchpad memory comes with a cost. Managing scratchpad memory by user is cumbersome and also error-prone. It also requires rewriting of existing application to make use of scratchpad memory. For the above mentioned reasons, there has been an extensive amount of work for automatic content selection into scratchpad memory in the past. In the following, we shall briefly describe the trend of research in this direction in past and how our work differs from the previous approaches.

3.6.4 Scratchpad allocation techniques

Scratchpad allocation can be *static* (where the content of the scratchpad is decided at the compile time and cannot be changed at runtime) or *dynamic* (where the scratchpad can be overwritten and reloaded at runtime). Both have its own advantages and disadvantages. Static scratchpad allocation schemes do not encounter any reloading cost at runtime. However, static allocation also limits number of variables to be allocated into scratchpad compared to dynamic scratchpad allocation. A significant amount of research efforts have been made [53; 54; 55; 56] for developing efficient scratchpad allocation schemes that aim for reducing average-case execution time (ACET) of a program. These works mainly takes a memory access profile of the program and try to optimize the most frequently accessed path. Our aim is to optimize worst case performance and since most frequently accessed path is not necessarily the worst case path, none of the before mentioned techniques can directly be applied for our purpose. Another inherent problem with any worst case performance optimization is that the worst case path may change after an optimization pass. As an example, if a variable is allocated into scratchpad accessed in the current worst case path π, π may not remain the worst case path after the allocation, as the cost of π has been reduced. Therefore, WCET analysis is carried out again to find the new worst case path π' to employ the next optimization pass. The approach proposed in [57] first addresses the problem of scratchpad allocation targeting towards the reduction of WCET. The work of [57] proposed different schemes of static scratchpad allocations for reducing the WCET of a program. An ILP-based approach was proposed, which is *optimal* (in the sense achieving the minimum WCET) but it cannot take into account of certain infeasible paths in a program. A greedy heuristic for scratchpad allocation was also proposed, which was shown to achieve nearly same performance gain with the optimal scheme. Additionally, the greedy heuristic can also take into account the infeasible path information. The approach proposed in [57] is based on static scratchpad allocation. Therefore, the entire content of the scratchpad is decided before the program execution and the content is *never* changed. Static scratchpad allocation has the advantage of avoiding any reloading cost of scratchpad at the runtime, however, it may suffer from poor scratchpad space utilization. As a result, dynamic scratchpad allocation for WCET reduction has also been investigated in [58].

The work of [57] and [58] address the allocation of data objects into scratchpad. It is worth mentioning that allocating program code into scratchpad requires additional care to maintain

program flow, while allocating program data into scratchpad generally calls for specific considerations depending on the type of the data (global, stack, or heap) and the different nature of their access. Allocation of code objects in scratchpad has also been addressed previously in [59] for ACET reduction and has recently been addressed in [60] for WCET reduction.

Scratchpad sharing among different processing elements in multiprocessor system-on-a-chip (MPSoC) has also been explored by researchers. Scratchpad allocation framework for average case performance improvement has been presented, among others, in [61] and [62]. Therefore, these techniques are not useful for improving the worst case performance of an application.

Static scratchpad allocation strategies for concurrent embedded software have recently been studied in [63] for worst case performance improvement. Scratchpad space is shared among multiple tasks through *overlay* if their execution times do not overlap. On the other hand, whether the execution time of two tasks overlap, depends on the allocated memory blocks from these two tasks into scratchpad. Because of this dependency, an iterative framework has been proposed in [63]. The iteration stops when there is no more change in the interference of execution times among all the tasks. However, there are two key limitations in [63]. First, [63] ignores the waiting time to access the shared bus. Secondly, the architecture explored in [63] only has a private scratchpad for each processing element and the private scratchpad is shared among different tasks by partitioning or overlay. Current commercial processors such as Cell [64] allow for the scratchpad space to be (virtually) shared among all the available processing elements through *local* or *remote* access. Our work on scratchpad allocation aims to optimize the Worst Case Response Time (WCRT) of an application by accurate content selection and *overlay* in this shared scratchpad space, by accounting for the variable bus delays. Chapter 10 describes our optimization framework in details.

Chapter 4

Unified Cache Modeling for WCET Analysis and Layout Optimizations

Shared resources are employed even in single core architectures. Unified cache is the most common form of resource sharing available in current generation processor chips. In real processors (such as Intel x86), the most common cache architecture is a multi level one. In the first level (L1), there are separate instruction and data caches. In the second level (L2), there is a unified cache which houses both instruction and data. Instruction accesses are looked up first in the L1 instruction cache, followed by the L2 unified cache, and finally in the main memory. Similarly, data accesses are looked up first in the L1 data cache, followed by the L2 unified cache and finally in the main memory. Therefore, we start the work in our dissertation by modeling this variety of resource sharing in single core — namely the unified cache.

4.1 Technical Contributions

In terms of technical contributions, ours is the first work to model the timing effects of a L2 unified cache which houses instruction as well as data. Previous works had either considered instruction cache or data cache but not both. In this work, we integrate the modeling of instruction and data accesses by modeling the timing effects of a unified (instruction + data) cache. Using our cache modeling, we can identify the different sources of WCET over-estimation in a multi-level cache architecture with instruction and data caches. Our cache modeling framework has been integrated into the open-source WCET analyzer Chronos [23]. Using our WCET analysis, we develop heuristics to perform *simultaneous* code and data layout optimizations which

can help reduce the WCET estimate. Previous works on WCET-driven compiler optimizations [50; 1] have studied code layout separately without concern for data layout, and this is problematic in the presence of a unified cache. Thus, we present a cache modeling framework which goes beyond existing approaches, integrate it into a state-of-the-art WCET analyzer, and use the analyzer results to guide novel WCET-driven layout optimizations.

4.2 Assumptions

We consider a memory hierarchy containing L1 instruction cache, L1 data cache and a L2 unified D/I cache. For simplicity, all our examples and evaluation assume that the cache replacement policy is LRU and the write policy is write-through with allocate, although the proposed analysis is not tied to a specific cache replacement or write policy. We also assume the following.

1. A piece of information is searched in the level 2 cache if and only if a cache miss occurs in level 1 cache. Cache of level 1 is searched always.

2. Every time a cache miss occurs in level L cache, the entire cache line containing the missing piece of information is loaded in cache of level L.

3. There is a separation of address space for instruction and data. That is from the memory address alone, it can be verified whether it is the address of an instruction or the address of a data.

4. Effects of micro-architectural features such as out of order pipeline, branch prediction (in particular timing anomalies created by the interaction of cache with these other features) are disregarded.

Assumption 1 rules out architectures where cache levels are searched in parallel to speed up the search for a piece of information. Assumption 2 rules out architectures with exclusive caches. These two assumptions also appear in [65].

4.3 Overview of our cache analysis

An overview of our cache analysis is shown in Figure 4.1. A separate address analysis, which predicts the range of data addresses accessed by each load/store instruction, is needed for data

cache analysis. From the result of L1 instruction and data cache analysis, the "$Compute\ CAC$" block in the diagram computes the access criteria for the unified cache which is used by the final analysis to compute the *hit/miss* classification of data and instruction in the same. All cache analysis results are used for the final WCET estimation.

To illustrate unified cache analysis consider the following code fragment and its corresponding assembly code targeting Simplescalar Portable Instruction Set Architecture (PISA).

```
int a[4][18];
for(j = 0; j < 4; j++)
 for(i = 0; i < 18; i++)
   a[j][i] = a[j][i] + 10;
```

```
00400208  addu      $4,$0,$0
00400210  addu      $3,$0,$5
00400218  lw        $2,0($3)
00400220  addiu     $4,$4,1
00400228  addiu     $2,$2,10
00400230  sw        $2,0($3)
00400238  addiu     $3,$3,4
00400240  slti      $2,$4,18
00400248  bne       $2,$0,00400218
00400250  addiu     $5,$5,72
00400258  addiu     $6,$6,1
00400260  slti      $2,$6,4
00400268  bne       $2,$0,00400208
```

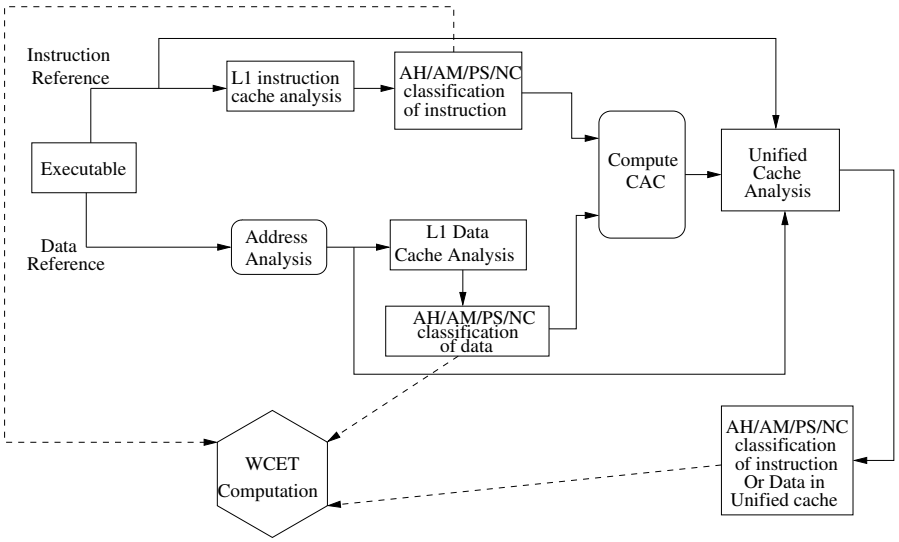


Figure 4.1: Overview of Cache Modeling Framework

Assuming that each integer takes 4 bytes to store and cache block size being 32 bytes, array a accesses nine memory blocks in the full computation when a starts from the memory block

boundary. Also assuming that each instruction take 8 bytes to store, the loop accesses four memory blocks for fetching instructions say $\{I_1, I_2, I_3, I_4\}$. Let $\{m_1, ..., m_9\}$ are the memory blocks accessed by a. Assume a direct mapped L1 data cache and for the sake of illustration let's say m_1 maps to the same cache block as m_9 in L1 data cache. Apart from these there are no other conflicts in data cache. Since access patterns are not considered, a persistence analysis on data cache cannot classify any of the accesses of the array a in the loop to be $persistent$ which leads to adding the cache miss penalty for all $4 \times 18 \times 2 = 144$ array accesses in the source code. But in reality, only 3 memory blocks are accessed per outer loop iteration leading to a total miss count of 12.

Now consider the presence of a L2 unified cache (common in commercial processors such as Intel x86) whose size is four times bigger than the L1 data cache by increasing the associativity from 1 to 4. Increase of associativity is a reasonable assumption as cache associativity generally increases with hierarchy level. In this case, persistence analysis on unified cache will not evict m_1 for accessing the memory block m_9. The instruction memory blocks $\{I_1, I_2, I_3, I_4\}$ being contiguous map to different cache sets. Moreover for a set-associative cache, one of these instruction memory blocks can co-exist with m_1 in a cache set of the unified cache. Thus, persistence analysis on the unified cache can declare all accesses in the loop to be $persistent$ in the same. Since access to unified cache is much faster than accessing memory and access of array a is persistence in unified cache in this example, overall estimate in WCET will have much tighter result. Persistence analysis of the data cache and unified cache results at the start of the loop are shown in Table 4.1 where l_{evict} represents cache blocks which may be evicted from the cache and l_i represents the usual cache blocks. The different rows in Table 4.1 represent different cache sets. For the above example, L1 data cache analysis encountered a cache $thrashing$ scenario which leads to a much higher WCET than expected. The example also shows a scenario where the presence of unified cache does not make much difference in concrete execution but certainly analyzing the unified cache makes us predict a much tighter WCET estimate.

On the other hand consider a very large loop which cannot entirely fit into the instruction cache. As a result in a concrete execution of the loop, cache $thrashing$ will take place in presence of only instruction cache. But in presence of a unified cache in the memory hierarchy this problem may be resolved as unified cache is generally much larger than level 1 caches. For illustration suppose in the example I_1 and I_4 conflicts. Thus every iteration will encounter two instruction cache misses apart from cold misses. But presence of a unified cache will resolve

L1 Data Cache		Unified Cache				
l_0	l_{evict}	l_0	l_1	l_2	l_3	l_{evict}
$\perp$	$\{m_1, m_9\}$	$\perp$	$\{m_1, m_9\}$	I_1	$\perp$	$\perp$
m_2	ϕ	m_2	I_2	$\perp$	$\perp$	$\perp$
m_3	$\perp$	$\perp$	$\{m_3, I_3\}$	$\perp$	$\perp$	$\perp$
m_4	$\perp$	$\perp$	$\{m_4, I_4\}$	$\perp$	$\perp$	$\perp$
m_5	$\perp$	m_5	$\perp$	$\perp$	$\perp$	$\perp$
m_6	$\perp$	m_6	$\perp$	$\perp$	$\perp$	$\perp$
m_7	$\perp$	m_7	$\perp$	$\perp$	$\perp$	$\perp$
m_8	$\perp$	m_8	$\perp$	$\perp$	$\perp$	$\perp$

Table 4.1: Example of Persistence Analysis in Unified Cache, $\perp$ represents empty cache line

this by getting the relevant instruction block from unified cache. From the analysis result in Table 4.1 also we can see that in the presence of a larger unified cache we can resolve this problem since all of the instructions have become $persistent$ in unified cache. This constitutes an example where cache thrashing is avoided in *concrete* execution because of unified cache, and this will also be captured in the unified cache analysis.

4.4 Details of Cache Analysis

For the rest of the discussion we consider a set associative cache with associativity A and with a set of cache lines $L = \{l_1, l_2, ..., l_n\}$ in a single set. The memory store is considered as a set of memory blocks $S = \{s_1, s_2, ..., s_m\}$. An abstract cache set is a mapping $\hat{d} : L \Rightarrow 2^S \cup \perp$ where each cache line corresponds to a set of memory blocks and $\perp$ captures the situation where a cache line is empty. Let $\hat{D}$ represents the set of all abstract cache states. To model the LRU replacement policy it is assumed that the memory blocks in the cache set are ordered by increasing age.

Data Cache Analysis The output of *address analysis* is used in data cache analysis. A key difference between instruction and data references is that the address set for the latter may not be a singleton set (for example consider array references). As long as a data reference accesses a single memory block, the $update$ function for any data cache analysis remains same as that of instruction cache analysis; in this case it is definitely known which memory block is accessed and thus it can be brought to the abstract cache set. On the other hand, if number of memory blocks accessed is more than one, it is not definitely known which memory blocks are accessed in concrete execution as address analysis computes an over-approximation of the

actual addresses accessed. Thus for our analysis to be safe, the $update$ functions for different data cache analysis (may, $must$ and $persistence$) become different. The $persistence$ analysis for data cache is described in [11] although no experimental results were presented. $Must$ analysis for data cache is introduced in [26]. For a precise analysis result of the unified cache in our architecture, we also need to perform may analysis on data cache. May data cache analysis classifies $all\text{-}miss$ data references of a program. As memory blocks corresponding to $all\text{-}miss$ data references in one cache level are always searched for in the next cache level, these memory blocks are potential candidates to be brought into the unified cache at level 2. We describe our proposed may analysis for data cache next.

May Analysis As described before, when the accessed memory block is a singleton, the $update$ function remains same as in the case of instruction cache analysis. But when number of accessed memory blocks is more than one, a $safe$ update function for may analysis should satisfy the following two properties:

1. All memory blocks possibly accessed by the address set must be brought into the abstract cache set and have lowest possible age in the corresponding set.

2. Age of all memory blocks that are already in the abstract cache and possible accessed must be decreased to the lowest possible age.

Thus we use the following generalized $update$ function for may data cache analysis:

$$\boxed{\hat{U}_{may}(\hat{d}, M) = \sqcup_{m_i \in M} \hat{U}(\hat{d}, m_i)} \tag{4.1}$$

Here M is the set of memory blocks accessed by the data reference, $\hat{U}_{may} : \hat{D} \times 2^S \rightarrow \hat{D}$ is the update function used for may data cache analysis, $\hat{U}$ is the $update$ function used in instruction cache analysis, $\hat{d}$ is the current data cache set and $\sqcup$ is the join operation used for may analysis. Informally, a may join operation is performed for each possible memory blocks accessed by the reference. It is clear that this update operation satisfies both of the above specified conditions of may analysis. Join operation for may data cache analysis remains same as that of may analysis for instruction cache [9].

A particular data access at some program point p is classified as $all\text{-}miss$(AM) if the abstract cache contains none of the memory blocks accessed by it at p. Otherwise the data access is categorized as $not\text{-}classified$(NC).

Following example shows the difference of $must$, may and $persistence$ analysis on data cache. Let an abstract cache set be as shown in the first row of Table 4.2 at some program point p. For a particular memory reference r, assume the address analysis module computes a range of addresses which corresponds to a set of memory blocks $M \subseteq S$. Let $\{s_x, s_y, s_z\} \subseteq M$ map to the same cache set whose abstract state is shown at row 1 of Table 4.2. Abstract cache sets for $must$, may and $persistence$ analysis after memory reference r are shown in subsequent rows.

$state$	l_1	l_2	l_3	l_4	l_{evict}
$Initial$	s_x	s_p	$\perp$	$\perp$	$\perp$
$Must$	$\perp$	$\perp$	s_x	s_p	$\perp$
May	$\{s_x, s_y, s_z\}$	s_p	$\perp$	$\perp$	$\perp$
$Persistence$	$\perp$	$\perp$	$\{s_x, s_y, s_z\}$	s_p	$\perp$

Table 4.2: Illustration of data cache analysis, $\perp$ represents empty cache line

Cache access classification Multi level instruction caches have been analyzed for WCET analysis in [66], and more recently in [65] which discusses timing anomalies permitted by previous approaches. In the presence of multi level cache hierarchy, a specified cache level may not be accessed at all. Thus the access categorization of a specified cache hierarchy must also be known. This categorization was first presented in [65]. For a given memory access r, CAC (cache access classification) of a particular cache level L can be as follows:

1. A: This means that the cache level L will always be accessed. For example for cache level 1 this is always true.
2. N: This means that the cache level L will never be accessed.
3. U: This means the access of this cache level L cannot be determined statically for this memory access.

CAC of cache level L for memory reference r is determined from the CAC and *hit-miss* categorization of r in cache level $L-1$. For example consider a two level hierarchy with L1 instruction cache, L1 data cache and a unified L2 cache. It is clear that AH categorized instructions or data are never brought into unified cache, as it is never accessed. On the other hand AM categorized instructions or data are always brought into unified cache as it is always accessed. For the other two categorizations it is not sure whether the unified cache is accessed or not. Thus all possibilities must be explored for a *safe* solution. The approach described in [65] has also been extended to analyze equivalent multi level data cache hierarchies in [67].

Unified Cache Analysis After $must$, may and $persistence$ analysis of instruction and data cache we have AH/AM/PS/NC classification of each instruction in the program and additionally if the instruction is a load/store instruction we also have the same classification for its data access. Moreover for each instruction and data access we know whether the unified cache will be accessed or not. Since level 1 caches are always accessed, only the *hit/miss* criteria of instruction and data access will decide the access classifications of unified cache.

Let an abstract cache set of unified cache be a mapping $\hat{u}_1 : L \Rightarrow 2^S \cup \perp$ where each cache line corresponds to a set of memory blocks. Let $CAC_u(i)$ represents the CAC (cache access classification as described in 4.4) of instruction i in unified cache and additionally if the instruction is a memory load/store $CAC_u(d_i)$ represents the CAC of data access at instruction i in unified cache.

Informally, for any memory block accessed, it is checked whether the corresponding access in unified cache is an A (always) access or not. If the unified cache has a N(never) classification for the same access, no update is performed. In an U(unknown) classification of the access, a join operation is performed on previous two possibilities depending on the kind of analysis (may, $must$ or $persistence$) and access type (instruction or data).

It is also worth mentioning that for each instruction in the program, instruction is fetched from memory or cache first and if the instruction is a load/store instruction then the data is fetched from memory or cache subsequently. Thus when updating the unified cache, we always update it first with the memory block representing the instruction and then with all memory blocks representing the data access (if any).

Algorithm 1 describes the operations carried out for each instruction i in unified cache analysis. Given an input abstract cache set $\hat{u}_1$ it produces the output abstract cache set $\hat{u}_f$ after the execution of instruction i.

In Algorithm 1, $\hat{U}$ and $\hat{U}_d$ represents the update functions used for instruction and data cache analysis respectively, and $\sqcup$ denotes the join function. Note that the update and the join function depends on the type of analysis performed (may, $must$, $persistence$). For our purposes, we performed both $must$ and $persistence$ analysis on unified cache. This allows us to categorize certain code/data accesses as AH — Always Hit (via must analysis) or PS — Persistent (via persistence analysis), thereby tightening our WCET estimate. We now discuss the experimental results obtained from our analysis.

Algorithm 1 Unified Cache Analysis. $\hat{u}_1$ is the input abstract cache state (for a cache set) and $\hat{u}_f$ is the output abstract cache state (for the same set) after executing a given instruction i.

Let m_i be the memory block corresp. to instruction i
if ($CAC_u(i) = A$) **then**
  $\hat{u}_m = \hat{U}(\hat{u}_1, m_i)$
else if ($CAC_u(i) = N$) **then**
  $\hat{u}_m = \hat{u}_1$
else if ($CAC_u(i) = U$) **then**
  $\hat{u}_m = \hat{U}(\hat{u}_1, m_i) \sqcup \hat{u}_1$
end if
if the instruction is not a load/store instruction **then**
  $\hat{u}_f = \hat{u}_m$
  return
end if
Let M is the set of data memory blocks accessed by instruction i
if ($CAC_u(d_i) = A$) **then**
  $\hat{u}_f = \hat{U}_d(\hat{u}_m, M)$
else if ($CAC_u(d_i) = N$) **then**
  $\hat{u}_f = \hat{u}_m$
else if ($CAC_u(d_i) = U$) **then**
  $\hat{u}_f = \hat{U}_d(\hat{u}_m, M) \sqcup \hat{u}_m$
end if

4.5 Analysis results

In this section we evaluate the accuracy and precision of our unified cache analysis. We have implemented the unified cache analysis inside the Chronos WCET analyzer framework [23]. To compare the overestimation of WCET we have taken four different cache configurations whose essential parameters are shown in Figure 4.2. For the rest of the discussion we shall use the abbreviations for cache configurations as shown in Figure 4.2. Our implementation has a 5 staged pipeline with in-order execution. Branch prediction is assumed to be perfect in all the experiments. L1 cache hit latency is 1 cycle and L1 cache miss penalty is 2 cycles. L2 cache miss penalty is 4 cycles. If there is no level 2 cache in the configuration, the cache miss penalty is taken to be 6 cycles. In Figure 4.2 all L1 caches have a block size of 32 bytes whereas all L2 caches have a block size of 64 bytes. For each of the cache configurations shown in Figure 4.2 we define two metrics, Sim and Est. Here Sim represents the observed WCET (in terms of cpu cycles) of a program and Est represents the WCET computed through static analysis (in terms of cpu cycles).

All experiments are run on a 3 GHz Pentium 4 machine having a 1 GB of RAM and running ubuntu Linux 8.10 operating system.

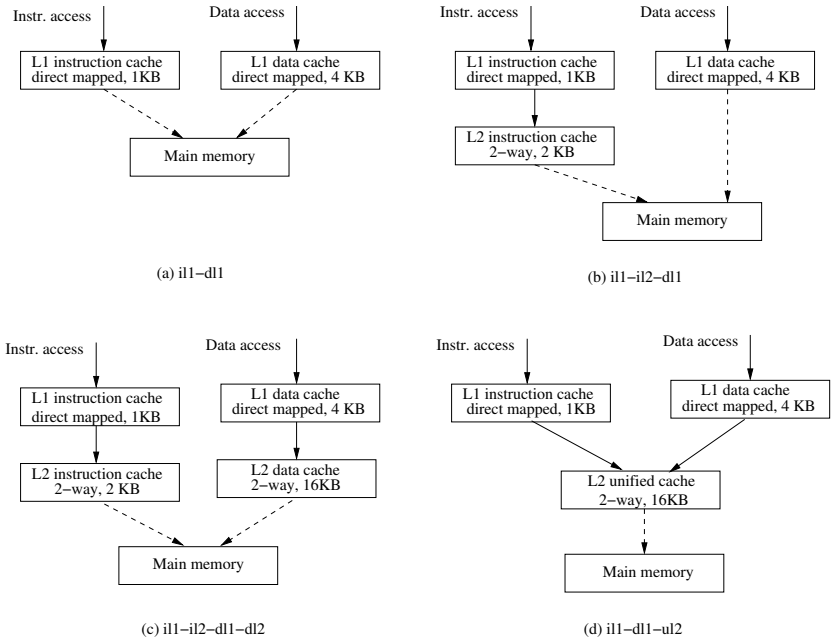


Figure 4.2: Different cache configurations used in experiments

Benchmarks We have used benchmarks described in table 6.1 from [2]. To test the effect of unified cache we have taken benchmarks having different characteristics as follows.

1. Benchmarks having small/medium loop (in terms of codesize) but accessing large amount of data (*e.g.* matmult,cnt,ns).

2. Benchmarks having large loop (in terms of codesize), and accessing large amount of data (*e.g.* fft, edn).

3. Benchmarks having large loop (in terms of codesize) but accessing small amount of data (*e.g.* fdct).

4. Benchmarks having small loop (in terms of codesize) and accessing small amount of data (*e.g.* qurt, expint).

We have benchmarks containing single as well as multiple paths. For example matmult, bsort100 are single path programs, whereas qurt,expint,fft have multiple paths.

Comparison of analysis precision Table 4.4 demonstrates the running time and accuracy of our analysis. Cache analysis time in Table 4.4 corresponds to the total time taken for multi-level cache analysis excluding the time for address analysis (which is presented separately). The WCET overestimation ratio for different cache configurations are shown in Figure 4.3.

Table 4.3: Description of Benchmarks used

Benchmark	*Description*	*Bytes*	*LOC*
matmult	Matrix multiplication of two 20 X 20 matrices	3737	163
cnt	Counts non-negative numbers in a 40 X 40 matrix	2880	267
bsort100	Bubblesort program	2779	128
insertsort	Insert sort a reverse array of size 10	3892	92
expint	Series expansion for computing an exponential integral function.	4288	157
bs	Binary search for the array of 30 integer elements.	4248	114
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample	11965	276
fdct	Fast Discrete Cosine Transform.	8863	239
fft	1024-point Fast Fourier Transform.	6244	135
ns	Search in a multi-dimensional array.	10436	535
qurt	Root computation of quadratic equations.	4998	166
edn	Implements the jpegdct algorithm together with other signal processing algorithms.	10563	285

Table 4.4: Accuracy and running time of WCET analysis for the different cache configurations described in Fig. 4.2.

	il1-dl1		il1-il2-dl1		il1-il2-dl1-dl2		il1-dl1-ul2		Analysis time	
Benchmark	*Sim*	*Est*	*Sim*	*Est*	*Sim*	*Est*	*Sim*	*Est*	Address analysis	Cache analysis
matmult	187056	360154	187000	360146	186985	224222	186985	224230	2.78	1.6
cnt	75278	103056	75232	103048	74432	83840	74432	83848	1.14	1.3
bsort100	2692	3471	2664	3463	2664	3347	2664	3347	1.01	1.1
insertsort	968	1509	936	1493	936	1341	936	1341	1.01	1.04
expint	2730	3491	2693	3491	2693	3487	2693	3487	1.05	1.22
bs	141	261	121	261	121	221	121	221	1.01	1.01
fir	348411	700311	348374	700303	348168	498027	348192	498051	1.01	1.3
fdct	2893	4300	2744	4276	2744	4276	2744	4276	0.01	1.2
fft	610117	900693	571309	900633	561984	596237	562345	745637	1.29	2.67
ns	7665	13053	7641	13053	7577	10409	7585	10409	1.01	1.08
qurt	1847	2910	1816	2902	1701	2846	1701	2846	0.2	1.6
edn	90730	99574	87945	98054	87945	91216	87945	96216	1.1	3

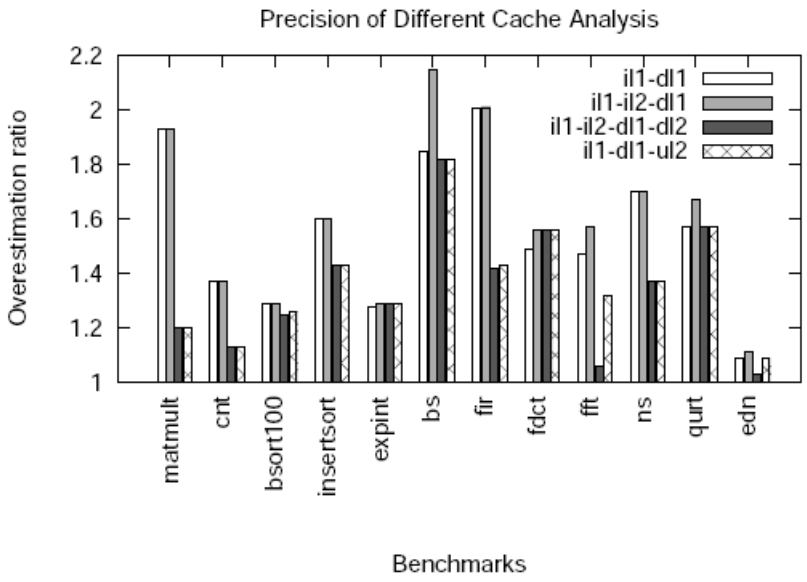


Figure 4.3: WCET overestimation for the cache configurations in Fig. 4.2

For data intensive programs (*e.g.* matmult, fir, ns), WCET estimates in presence of unified caches are much tighter than the WCET estimated in presence of only L1 data cache. Presence of a L2 data cache also reduces the WCET. For these data intensive programs, a large number of memory blocks whose *hit-miss* criteria were *not-classified* (NC) in L1 data cache, become *persistent* in unified cache or L2 data cache. We also observe that modeling only an L2 instruction cache together with the L1 instruction cache does not reduce the WCET significantly for these programs. The reason is all loops of these programs can fit into the L1 instruction cache. Thus no cache *thrashing* happens in L1 instruction cache when executing the loop body. Only reduction in WCET estimation by modeling the L2 instruction cache may come through the higher block size of the same. We also observe that the estimate with unified cache and separated L2 data cache are almost the same. This signifies that there is little interference between instruction and data in the L2 unified cache.

On the other hand, benchmarks which have very large loops in terms of codesize (*e.g.* edn), modeling only an L2 instruction cache shows significant improvement in WCET. There is one loop in edn which cannot fit entirely in a 1 KB instruction cache. Thus in presence of an L2 instruction cache, all instructions in the loop which would have been evicted from L1 cache, become $persistent$ in the L2 cache and reduces the overall WCET estimation. However there is a significant amount of data accesses in edn; some of which become $persistent$ in presence of a unified cache and reducing the WCET estimation even more.

For benchmarks which have very small loop size (in terms of codesize) as well as access very small set of data (*e.g.* qurt, expint, bsort100), the WCET estimate cannot be reduced much by modeling any type of L2 caches. The reason is, all loops as well as accessed data memory blocks for these benchmarks can fit in L1 instruction and data caches respectively and thus getting no significant reduction in WCET in presence of L2 caches.

Finally, we observe that WCET estimates in presence of separate L2 instruction and data caches are almost same for all of the benchmarks except edn and fft. For these two benchmarks, there is a significant amount of interference between instruction and data in the unified cache (for which the WCET estimate is increased) — an issue we discuss in the next section.

4.6 WCET-centric code and data layout

In this section, we shall describe how our unified cache modeling framework can be used to find out possible conflicts between instruction and data memory blocks. We then use such conflict information to change the layout of code and data simultaneously and thereby improving the overall cache performance of an application.

In the presence of unified caches, conflict misses may occur between instruction and data. Thus the WCET in presence of a L2 unified cache cannot be better than the WCET in presence of separate L2 data and L2 instruction caches of same size. However a unified cache reduces lot of storage cost. Thus if the code and data layout in the program are placed such that minimal conflict misses occur in unified cache, WCET/ACET of an application can be highly reduced.

Procedure positioning is a well known compiler optimization aiming at the improvement of instruction cache behaviour. A recent paper [1] has proposed procedure positioning optimizations driven by WCET information to effectively minimize the program's worst case behaviour. However in presence of unified caches, this problem becomes more challenging as instruction may interfere with data and vice versa. Thus there is a need of simultaneous change of code and data layout for WCET reduction in presence of unified caches. We present here a fast heuristic based and unified cache aware algorithm for simultaneously changing the code and data layout to effectively minimize the WCET of a program.

Issues with WCET-centric procedure positioning in presence of unified cache Current WCET-centric procedure positioning algorithm may not be helpful in presence of unified caches. For example let us consider the program below:

```
void f1()
{
  .....
  for(i = 0; i < N; i++) {
    if(...) f3();
    else {
     f2(); a[i] = a[i] + 10;
    }
  }
}
```

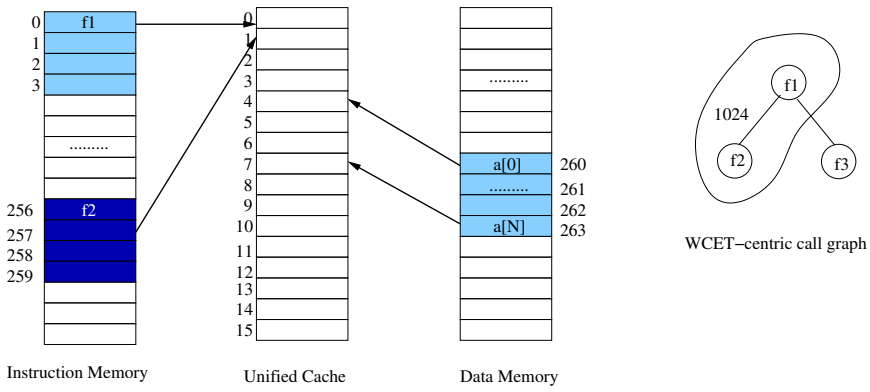


Figure 4.4: Code and data layout before procedure positioning

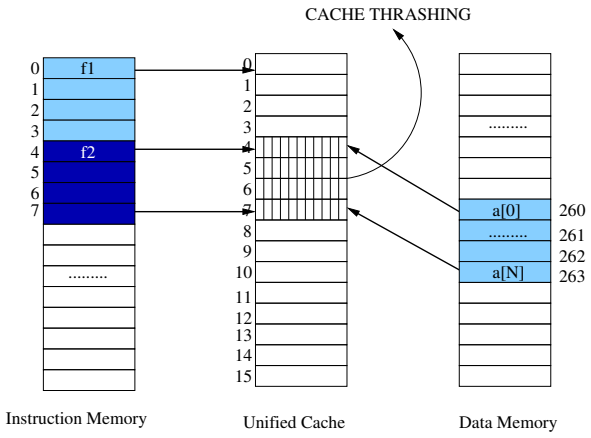


Figure 4.5: Code and data layout after procedure positioning by [1]

Instruction and data memory layout for procedures f1, f2 and array a are shown in Figure 4.4. For sake of illustration assume a direct mapped unified cache with 16 cache lines. Initially procedure f1 and f2 conflict each other in unified cache as shown in Figure 4.4. Array a maps

to cache lines 4 to 7. Because of the initial layout of f1 and f2, whenever f2 is called memory blocks corresponding to procedure f1 got evicted and when control comes back to f1 again, memory blocks corresponding to f2 are replaced eventually leading to a poor WCET estimate. Thus a WCET-centric procedure positioning algorithm as described in [1] re-position the procedures from a "WCET-centric call graph" so that most frequently called procedures are placed in contiguous memory locations. The WCET-centric call graph of a program[1] is computed by a WCET analyzer and is invariant of all program executions. Call frequency of each edge is computed by the WCET analyzer and not by profiling. The WCET-centric call graph of the example program is shown in Figure 4.4. The marked portion of the call graph corresponds to the worst case path and edges corresponding to the worst case path is labeled with call frequencies computed by the WCET analyzer. The layout after procedure positioning is shown in Figure 4.5. It is clear that without having the knowledge about where array a was mapped, procedure positioning in a unified cache leads to a layout which may encounter cache thrashing scenario as shown in Figure 4.5. This leads to the motivation of changing the layout of instruction and data simultaneously in presence of unified cache which we are going to describe next.

Simultaneous procedure and data positioning The idea behind simultaneous procedure and data positioning is to consider a unified memory and apply the general positioning algorithm.

WCET-centric unified graph We define a unified undirected graph $G_{uni} = (V, E)$ which is an extension to the call graph. We have $V = P \cup R$ where P is the set of nodes corresponding to all procedures and R is the set of nodes corresponding to all data references in the program. There is an edge $e \in E$ between $p_1 \in P$ and $p_2 \in P$ if p_1 calls p_2. Similarly there is an edge $e \in E$ between $p_1 \in P$ and $r_1 \in R$ if r_1 is inside procedure p_1. An edge $e \in E$ can be between two data references r_1 and r_2 if and only if $references(r_1) \cap references(r_2) \neq \phi$ i.e. two data references access some common memory blocks. Execution frequencies of all edges of our unified graph are computed from the WCET analyzer. Clearly edges between two data reference nodes do not have any associated frequency, they are drawn only to capture the overlapping memory access behavior. The unified graph for the example in Figure 4.4 appears in Figure 4.6 and the worst case path is marked. Edges belonging to the worst case path are labeled with execution frequencies. Data references $r1$ and $r2$ represent two references to array

[1]The nodes of a call graph denote procedures, and edges denote calling relationships. The edges are typically weighted with call frequencies.

a (one for load and another for store).

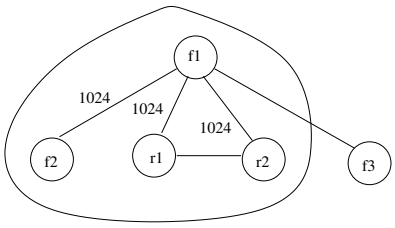


Figure 4.6: WCET-centric unified graph

Algorithm Our technique for simultaneous data and procedure positioning is described in Algorithm 2. In the algorithm, the $maxEdge$ function selects an edge $e \in E$ labeled with highest execution frequency at each step. If any node representing the edge (returned by end_1 and end_2 functions) is a data reference, then all overlapping data reference nodes with the current one are first merged together by $collapseData$ function to form a super-node. Subsequently the $collapseEdge$ function collapses the edge selected in that step to form a single node. This function also modifies all related execution frequencies. If none of the ends of a selected edge is a data reference node, only the $collapseEdge$ function is called to form a super-node as overlapping data references (if any) are already captured in the existing nodes. The algorithm terminates when no edges are left in the WCET-centric unified graph. After the graph has been merged to a single node, the layout is computed assuming the presence of a single unified memory and shifting to other memory such that mapping to the unified cache line is preserved.

Algorithm 2 Simultaneous code and data positioning. G_{uni} is the unified graph and R is the set of nodes in G_{uni} which represent data references.

```
repeat
  e = maxEdge(G_uni);
  if (e = φ) then
    return;
  end if
  if (end_1(e) ∈ R or end_2(e) ∈ R) then
    collapseData(e);
  end if
  collapseEdge(e);
until false
```

The collapsing of the unified graph in Figure 4.6 is depicted in Figure 4.7. For our example, $f1$ and $f2$ are allocated contiguously in instruction memory. Memory blocks corresponding to $r1$ and $r2$ are allocated in data memory such that they will map to the same cache line as if it would have been allocated contiguously after $f1$ and $f2$ assuming a hypothetical unified

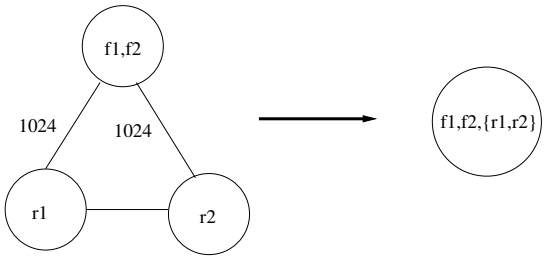


Figure 4.7: Transforming the unified graph of Figure 4.6

memory. The layout produced by our method is shown in Figure 4.8. As pointed out before our algorithm is unified cache aware i.e. assumes the knowledge of unified cache and also assumes the start of instruction and data memory.

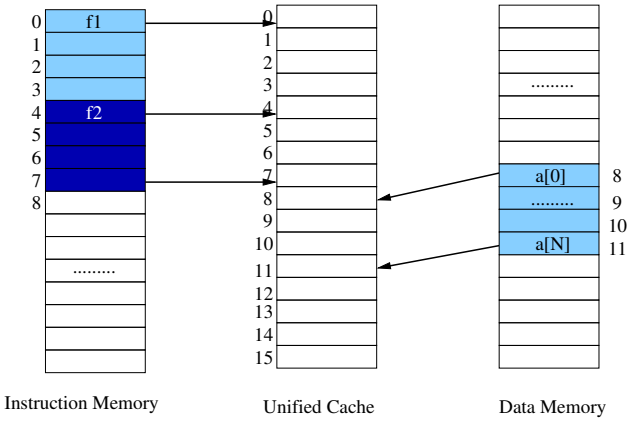


Figure 4.8: Final layout after our code + data positioning

Experiments To evaluate our heuristic, we compared the procedure positioning method of [1] with our unified code and data layout method. We chose the two benchmarks in our benchmark-suite which have large codesize as well as manipulate large amounts of data. These two benchmarks are fft and edn. We measure the amount of WCET reduction due to the procedure positioning method of [1] as well as our unified code and data layout method for these two benchmarks. A multi-level cache architecture with a L2 unified cache is assumed (see Fig. 4.2(d)). The results appear in Table 4.5. Est_{ul_2} represents the WCET estimate (in presence of a unified L2 cache) assuming a default layout for code/data (*e.g.* code is laid out as it appears in the program). Est_p denotes the WCET estimate using the procedure positioning of [1] and Est_{p+d} captures the WCET estimate using our simultaneous procedure/data positioning.

As expected, Table 4.5 shows that procedure positioning heuristic of [1] alone is not useful in presence of unified caches. But by simultaneous data and procedure positioning we are able to reduce the WCET estimate by 3% for edn and almost 18% for fft. For fft benchmark we

Table 4.5: Reduction in WCET estimates via change in layout

Benchmark	Est_{ul_2}	Est_p	Est_{p+d}
edn	96216	97240	93108
fft	745637	745638	608418

observe that using unified caches at level 2 increases the WCET by almost 0.15 million cycles. fft has fairly large loop structures (in terms of codesize) and it is a data intensive program. In case of improper data and instruction layout there is sufficient number of conflict misses in unified cache which may give a poor WCET estimate. Thus by applying our heuristic we are able to bring down the WCET estimate for fft substantially.

4.7 Chapter Summary

In this Chapter, we have developed a cache modeling framework for Worst-case Execution Time (WCET) analysis of real-time embedded software. Our framework considers a generic multi-level cache architecture with separated instruction and data caches in the first level and a unified (code+data) cache in the second level. Unified cache is the most common in commercial processors such as Intel x86 and ARM. Existing works on cache modeling have so far considered either instruction or data caches but not both. Our experiments indicate that our analysis of the multi-level unified cache architecture produces tight WCET estimates with low running time overheads.

We also exploit our WCET analysis of the unified cache to build WCET-centric compiler optimizations. In particular, we develop a joint (code + data) layout heuristic which leads to better timing predictability in the presence of a unified cache as compared to existing WCET-centric code positioning methods. These methods are oblivious of the data layout, whereas our joint layout is aware of the unified cache. As a result, our combined code and data layout achieves greater WCET reduction in the presence of a unified cache.

Chapter 5

Modeling Shared Cache for Timing Analysis

In the previous chapter, we have seen the timing unpredictability arising due to the presence of unified cache. Even for a single thread of execution, a unified cache may introduce difficulties in WCET analysis due to the sharing of different instruction and data memory blocks. However in a single thread of execution, the access sequences of instruction and data memory blocks are mostly predictable (since the instruction is always fetched before the same instruction accesses a data value). On the other hand, the modeling of shared caches in multi-core poses more difficulties, as the thread interleaving pattern is non-deterministic and in general, it is infeasible to enumerate all thread interleaving patterns. In this Chapter, we present a novel shared cache modeling framework which significantly improves the analysis precision over the state-of-the-art shared cache modeling.

5.1 Introduction

Recall that WCET estimation usually involves a program level path analysis (to determine the infeasible paths in the program's control flow graph) and micro-architectural modeling (to accurately determine the maximum execution time of the basic blocks). Micro-architectural modeling usually involves systematically considering the timing effects of performance enhancing processor features such as pipeline and caches. Cache analysis for real-time systems is usually accomplished by abstract interpretation. This involves estimating the cache behavior of a basic block B by considering the incoming flows to B in the control flow graph. The memory accesses

of the incoming flows are analyzed to determine the cache hits/misses for the memory accesses in B. Since programs contain loops, such an analysis of memory accesses involves an iterative fixed point computation via a method known as abstract interpretation. Abstract interpretation is usually efficient, but the results are often not precise. This is because the estimation of memory access behavior are "joined" at the control flow merge points - resulting in an over-estimation of potential cache misses returned by the method.

In this Chapter, we develop a cache analysis framework which improves the precision of abstract interpretation, without appreciable loss of efficiency. We augment abstract interpretation with a gradual and controlled use of model checking, a path sensitive search based formal verification method. Because of path sensitivity in its search - model checking is known to be of high complexity. Hence abstract interpretation based analysis cannot be naively replaced with model checking for analysis of cache behavior. Recent works [68] which have advocated combination of abstract interpretation and model checking for multi-core software analysis - restrict the use of model checking to program path level; cache analysis is still accomplished only by abstract interpretation. Indeed almost all current state-of-the-art WCET analyzers (such as Chronos [23], [24]) perform cache analysis via some variant of abstract interpretation. Model checking is usually found to be not scalable for micro-architectural analysis because of the huge search space that needs to be traversed. The main novelty of our work lies in integrating model checking with abstract interpretation for timing analysis of cache behavior.

Our baseline analysis is abstract interpretation. Potential cache conflicts identified by abstract interpretation are then subjected to model checking. Our goal is to rule out "false" cache conflicts which can occur only on infeasible program paths. Such false conflicts are considered by abstract interpretation since its join operator (which merges the estimates from paths at control flow join points) conservatively considers all possible cache conflicts on any path in the control flow graph. The path sensitive search in model checking naturally rules out the infeasible program paths and the cache conflicts incurred therein.

One appealing nature of our analysis method is that the results are always safe. We start with the results from abstract interpretation and gradually refine the results with repeated runs of model checking. Model checking is a property verification method which takes in a system/program P and a temporal logic property φ, where φ is interpreted over the execution traces[1] of P. It checks whether all execution traces of P satisfy φ. Given a potentially conflicting pair of

[1]We consider only Linear Time Temporal Logic properties here.

memory blocks, we can model check a property that the pair never conflicts in any execution trace of the program. If indeed the conflict pair is introduced due to the over-approximation in abstract interpretation - model checking verifies that the conflict pair can never be realized. We can then rule out the cache misses estimated due to the conflict pair and tighten the estimated time bounds.

The property checked in a single run of model checking involves certain cache conflicts identified by abstract interpretation - model checking then verifies whether these conflicts are indeed realizable. Thus, the scalability of our framework is never in question. Given a time budget T, we can first employ abstract interpretation and then employ as many runs of model checking as we can within time T. Of course, given more time, the results are more precise.

We finally show that such a compositional cache analysis framework is generic in nature and the use of model checking can be replaced by different other forms of property checking methodologies, such as constraint solving. Constraint solving technology has made significant progress with the advances in *satisfiability modulo theory* (SMT). A constraint solver can be used to explore different feasible program paths. Such a constraint solver based path exploration executes the program based on the *symbolic* input variables (termed as *symbolic execution* in the literature). Given a formula φ to check at a particular program location, a constraint solver is used to check the satisfiability of φ whenever the same program location is visited during the symbolic execution. The feasibility of a path is checked *on-the-fly* during the execution by sending a query to the SMT based constraint solver. Due to this inherent path sensitive nature of symbolic execution, the spurious cache conflicts can be eliminated when they are introduced due to the over-approximation of abstract interpretation. As the SMT technology is continuously evolving, we believe that the composition of abstract interpretation and constraint solving gives another exciting opportunity for WCET analysis.

Technical contribution In summary, we present a generic cache analysis framework based on abstract interpretation, model checking and constraint solving. Depending on the time budget for analysis and the analysis precision required - the framework can be tuned to analyze cache hit/miss classifications for timing analysis. Our experimental results on the moderate to large scale WCET benchmarks [2] show substantial improvement in the precision of multi-core timing analysis results with limited time overheads. This yields a parameterizable cache analysis framework for real-time systems which is generic, precise and scalable.

5.2 A background on existing cache analysis

WCET analysis of a single task WCET analysis of a single task is broadly composed of two different phases: i) micro-architectural modeling and ii) path analysis. Micro-architectural modeling analyzes the timing characteristics of different hardware components (*e.g.* cache, pipeline, branch predictor) and works at the granularity of basic blocks. As an outcome of micro-architectural modeling, we obtain the WCET of each basic block in the examined program. On the other hand, path analysis uses the WCET of each basic block as input and searches for the *longest feasible program path*. Our baseline implementation employs the separated cache and path analysis as proposed in [9]. [9] uses abstract interpretation (AI) for cache analysis and integer linear programming (ILP) for path analysis. We assume *least recently used* (LRU) cache replacement policy. We implement *must* and *may* cache analysis to classify memory blocks as *all-hit* (AH) and *all-miss* (AM) respectively. *Must* analysis is used along with virtual inline and virtual unrolling (VIVU) as discussed in [9]. In VIVU approach, each loop is unrolled once to distinguish the *cold cache misses* at first iteration of the loop. AH categorized memory blocks are always in cache when accessed. On the other hand, AM categorized memory blocks are never in cache when accessed. If a memory block cannot be classified as either of two (AH or AM), it is considered *unclassified* (NC). Cache analysis outcome is used for computing the WCET of each basic block. Finally, longest path search in a program is formulated as an integer linear program. The formulated ILP uses the basic block WCETs and structural constraints imposed by program control flow graph (CFG). Infeasible program path informations are also encoded as separate ILP constraints using the technique explored in [22]. The solution of the formulated ILP returns the whole program WCET.

Inter-core cache conflict analysis Inter-core cache conflict analysis computes the conflicts generated in shared cache. Conflicts in shared cache, on the other hand, are generated by the tasks running on different cores. Till now, only a few solutions have been proposed for analyzing timing behaviors of shared cache [15; 37; 35]. However, all of them suffer from over-estimating the inter-core cache conflicts. We use our former work on shared cache analysis [15], which employs a separate shared cache conflict analysis phase. Shared cache conflict analysis may change the categorization of a memory block m from all-hit (AH) to unclassified (NC). This analysis phase first computes the number of unique conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially

replace m from shared cache. More precisely, cache hit/miss categorization (CHMC) of m is changed from all-hit (AH) to unclassified (NC) if and only if the following condition holds:

$$N - age(m) < |\mathcal{M}_c(m)| \tag{5.1}$$

where $|\mathcal{M}_c(m)|$ represents the number of conflicting memory blocks from different cores which may potentially access the same L2 cache set as m. N represents the associativity of shared L2 cache and $age(m)$ represents the *age* of memory block m in shared L2 cache set in the absence of inter-core conflicts. Therefore, $N - age(m)$ specifically represents the amount of shift that memory block m can tolerate before being replaced from the cache. We call the term $N - age(m)$ as *residual age* of m.

5.3 Our proposed analysis framework

5.3.1 General framework

Figure 5.1(a) demonstrates the general analysis framework. Our goal is to refine the abstract interpretation (AI) based cache analysis through model checking (MC). *Cold cache misses* are unavoidable and AI based cache analysis can accurately predict the set of cold cache misses. However, AI based cache analysis suffers from overestimating the *conflict misses* in a cache. With the advent of multi-core architectures, it has become important to precisely estimate the timing behaviour of shared cache. AI based shared cache analysis suffers from precisely estimating the inter-core cache conflicts, which is generated in the shared cache by a task running on a different core. Figure 5.1(b) pictorially represents the inter-core cache conflicts generated in the shared cache.

Even though the basic goal of our framework is cache conflict refinement, the notion of cache conflict may vary depending on the outcome of AI based cache analysis. For example, in inter-task cache conflict refinement, initial CRPD analysis produces a set of ECBs, which can be considered as the set of cache conflicts. On the other hand, during intra-task and inter-core cache conflict refinement, we get the cache hit miss classification (AH, AM or NC) of each memory block. A memory block might be categorized as NC due to its conflicts with more than one memory block. Therefore, by refining one NC categorized memory block into AH, we may reduce more than one cache conflict pairs, which in turn results in an improvement of WCET.

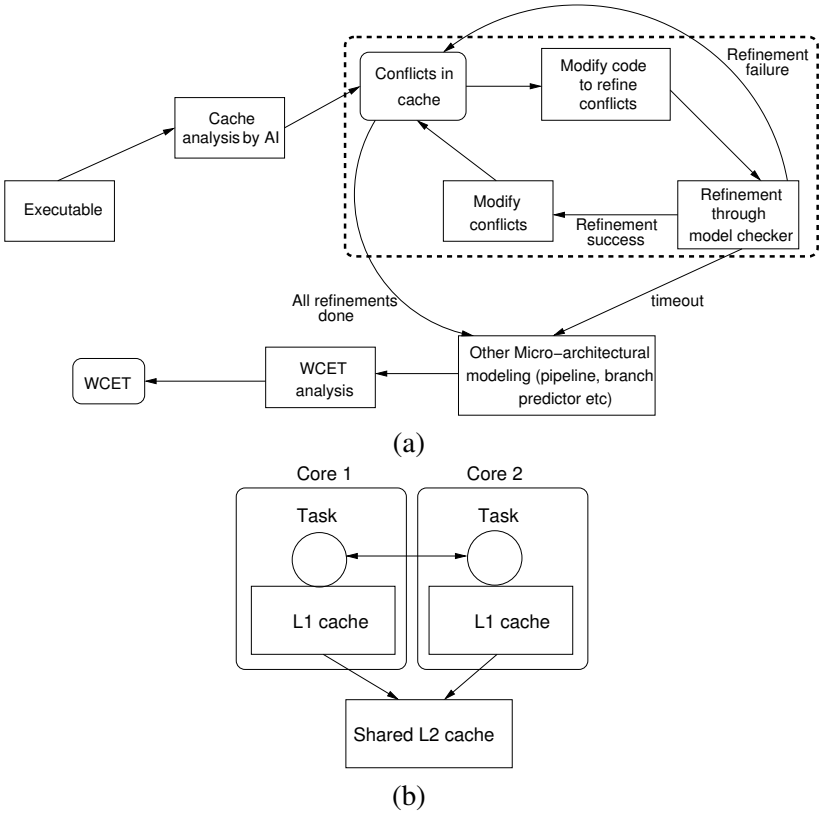


Figure 5.1: (a) inter-core cache conflicts, (b) General framework of our WCET analysis which combines abstract interpretation and model checking

In Figure 5.1(a), the dotted boxed portion captures the shared cache conflict refinement. The refinement of cache conflicts is iteratively performed through model checking on a modified program. We rule out the cache accesses for which AI has generated precise information. Therefore, the model checker refinement phase works on a very small subset of all cache accesses. The iterative refinement through model checking eliminates several infeasible paths from the candidate program, resulting in the removal of several unnecessary conflicts generated in a particular cache set. The iterative refinement is continued as long as the time budget permits or all possible refinements have been performed by MC. Recall that the WCET analysis process can broadly be categorized into two phases: micro-architectural modeling and path analysis. The infeasible path exploration by the model checker is only performed for refining cache conflicts (*i.e.* during the micro-architectural modeling phase). For path analysis, our framework encodes the infeasible path information as separate ILP constraints (for details, refer to [22]). Infeasible path constraints are finally used in the global ILP formulation for computing WCET. There are two important advantages of our framework: first, the iterative MC refinement can be terminated at any point if the time budget exceeds. The resulting *cache conflicts*, after a partial

refinement, can *safely* be used for estimating the WCET. Secondly, our framework can be composed with other micro-architectural features (*e.g.* pipeline, branch prediction) and thereby, not affecting the flexibility of AI-based cache analysis.

5.3.2 A general code transformation framework

Any code transformation for refining various cache conflicts can be represented by a quintuple $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ as follows:

- $\mathcal{L}$: Set of conflicting memory blocks in the cache set for which the refinement is being made.
- $\mathcal{A}$: The property which need be checked by the model checker. The property is placed in form of an "assertion" clause, which validates $\mathcal{A}$ for all possible execution traces of the modified code.
- $\mathcal{P}_l$: Set of positions in the code where the conflict count would be incremented. These are the set of positions where some memory block in $\mathcal{L}$ might be accessed.
- $\mathcal{P}_c$: Position in the code where property $\mathcal{A}$ would be placed.
- $\mathcal{I}$: Set of positions in the code to reset conflict count. Recall that we consider LRU cache replacement policy. A memory block m becomes the *most recently used* immediately after it is accessed. Therefore, if we are counting cache conflicts with m, the conflict count must be reset after m is accessed.

Any model checker refinement pass corresponds to a specific cache set and therefore, conflicts are defined for a specific cache set in each code transformation. Consequently, computation of $\mathcal{L}$ and $\mathcal{P}_l$ depends only on the cache set for which the conflicts are being refined.

In subsequent sections, we shall describe the instantiation of the framework in Figure 5.1 for refining shared cache conflicts (as shown in Figure 5.1(b)). We shall also show how $\mathcal{A}$, $\mathcal{P}_c$ and $\mathcal{I}$ are configured for refining the inter-core cache conflicts.

For our subsequent discussions, we shall use the example in Figure 5.2. Parameter z can be considered as an input to the program. Control flow graph (CFG) of the loop body and the accessed memory blocks are also shown in Figure 5.2.

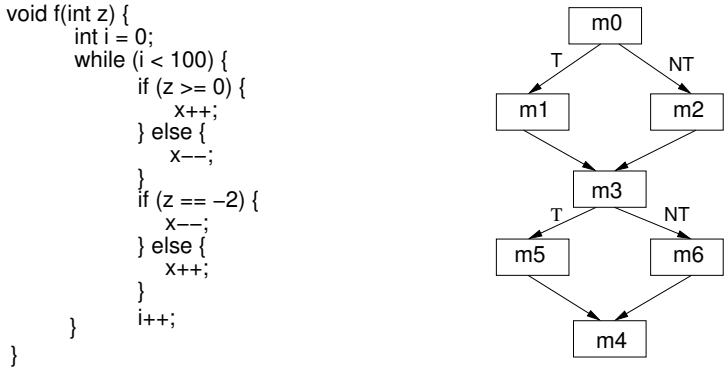


Figure 5.2: Example program and its corresponding control flow graph (CFG) without the backedge

5.3.3 Refinement of inter-core cache conflicts

We describe the refinement of inter-core conflicts generated in a shared cache (as shown in Figure 5.1(b)). Recall from Equation 5.1 that the precision of shared L2 cache analysis largely depends on the accuracy of estimating the term $|\mathcal{M}_c(m)|$. The model checking pass in our framework refines the set $\mathcal{M}_c(m)$ by exploiting infeasible paths in the conflicting task.

Figure 5.3 demonstrates the instantiation of our general framework for inter-core conflict refinement. We only target the memory blocks whose categorizations are changed from AH to NC in a shared cache conflict analysis phase. Consider such a memory block m mapping to an N-way associative shared L2 cache set i. Disregarding the inter-core conflicts, assume the *maximum LRU age* of m in cache set i is denoted by $age(m)$. Therefore, if the amount of inter-core conflicts (in cache set i) is bounded by $N - age(m)$, we can guarantee that m will remain a shared L2 cache hit, despite inter-core conflicts. Recall that $N - age(m)$ is called the *residual age* of m. Further assume t_c is a task which may generate inter-core cache conflicts and C_i serves the purpose of counting inter-core conflicts in shared L2 cache set i generated by t_c. Therefore, we use the model checker to verify an "assertion" property $C_i \leq N - age(m)$. Identical to inter-task cache conflict refinement, we need to check the total amount of cache conflicts generated by task t_c. Therefore, in our transformed code, we initialize C_i only once, before any cache blocks accessed by t_c and we check the "assertion" property just before the exit point of t_c.

The example in Figure 5.2, assume that $m1$ and $m5$ map to the same cache set of a 2-way set associative L2 cache. Further assume that we are trying to refine the inter-core cache conflicts generated to a task t' and t' is running in parallel on a different core with the task in Figure

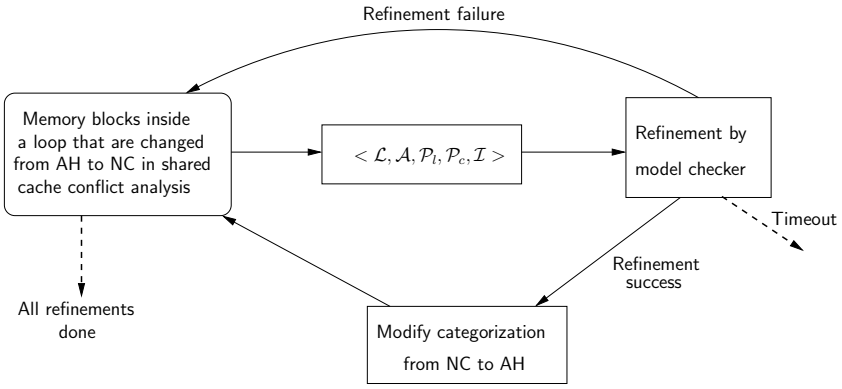


Figure 5.3: Refinement of shared cache conflict analysis

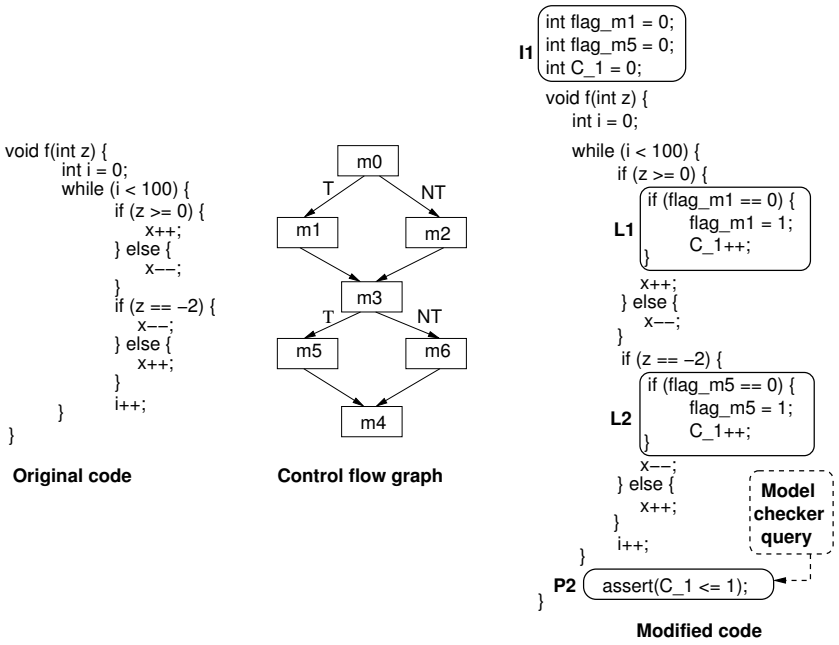


Figure 5.4: Inter-core cache conflict refinement

5.2. Consider t' accesses a memory block m', which map into the same shared L2 cache set as $m1$ and $m5$. Finally assume that m' is an all-miss (AM) or unclassified (NC) in L1 cache, but an all-hit (AH) in L2 cache *with residual age one*, in the absence of inter-core cache conflicts. Previous analysis will compute $|\mathcal{M}_c(m')|$ as 2 (due to $m1$ and $m5$ in the conflicting task). Since the residual age of m' is one, the categorization of m' will be changed to NC (Equation 5.1), leading to unnecessary conflict misses. We modify the code to check whether the number of unique inter-core conflicts is less than or equal to the residual age of m'. The transformation is similar to Figure 5.4 where C_1 serves the purpose of counting unique cache conflicts with m' in shared L2 cache. The model checker will satisfy the assertion P2 in Figure 5.4 due to the infeasible path $m1$-$m3$-$m5$. Consequently, we shall be able to derive that the amount of inter-

core conflicts with m' never exceeds the residual age of m'. Therefore, the categorization of m' is kept all-hit (AH). Configuration of our code transformation framework $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ is identical to the inter-task cache conflict refinement as follows: $\mathcal{L} = \{m1, m5\}$, $\mathcal{P}_l = \{L1, L2\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C\_1 \leq 1$, $\mathcal{P}_c = \{P2\}$ and $\mathcal{I} = \{I1\}$.

Although we show the transformation for a two core system, our framework does not have the strict limitation of working only for two cores. However, one model checker invocation can verify only one task. Therefore, to refine conflicts from X different tasks $t_1, t_2, \ldots, t_X$ running on X different cores, we first employ an additional *compose* phase in transformation. The *compose* phase sequentially composes $t_1, t_2, \ldots, t_X$ (in any order) into a single task T. The infeasible paths in any task $t_1, t_2, \ldots, t_X$ are preserved in task T. Consequently, our code transformation technique can be applied to T in exactly same manner as described in the preceding to refine conflicts from $t_1, t_2, \ldots, t_X$. Since the composition is sequential, number of conflicts are accumulated from all X cores. Model checker refinement passes can then be carried out on task T.

5.3.4 An extension to a generic cache analysis framework

In [69], we have shown that such a combination of abstract interpretation and model checking is generic in nature and it can be used to refine different varieties of cache analysis. More precisely, our framework proposed in Figure 5.1(a) can be instantiated for refining three different varieties of cache analysis: first, cache analysis in single core, secondly, cache analysis for multi-tasking system in single core and thirdly, shared cache analysis for multi-core systems. In the preceding, we only present the instantiation of our framework for shared cache analysis. For further details, readers are referred to [69].

5.3.5 Optimizations

To reduce the number of calls to model checker, we cache the verification results. Recall that the "assertion" property verified by the model checker was always placed at the end of conflicting task during inter-core cache conflict refinement. Therefore, the following optimization can be applied only during inter-core conflict refinement.

Model checker results are stored as a triple $(set,\ result_{mc},\ conflicts)$. The triple has the following meaning:

- set : Cache set for which the refinement is being made.

- $result_{mc}$: Returned result by the model checker. Assume $result_{mc}$ is one for a successful verification and zero otherwise.

- $conflicts$: Number of conflicts in the assertion property. If we verify an assertion property $C_i \leq N$, value of $conflicts$ is N.

In Figure 5.4, we store $(1, 1, 1)$ after the successful refinement (assuming $m1$ and $m5$ map to cache set 1). Assume any other assertion of form $C_{set'} \leq N'$ is needed to be verified, where set' is the cache set for which the conflicts are being refined. We search the cached results of form $(set,\ result_{mc},\ conflicts)$ and take an action as follows:

- $set = set' \wedge result_{mc} = 0 \wedge N' \geq conflicts$: Assertion failure is returned. If the refinement previously failed for a less number of conflicts, it will definitely fail for more conflicts.

- $set = set' \wedge result_{mc} = 1 \wedge N' \leq conflicts$: Assertion success is returned. If the refinement was previously satisfied for more number of conflicts, it must be satisfied for less number of conflicts.

If none of the entries satisfy the above two conditions, a new call to the model checker is made. Depending on the outcome, the new result is cached accordingly for future use.

5.4 Implementation and evaluation using CBMC

5.4.1 Implementation

We have used the Chronos timing analysis tool [23] in which we have already integrated the AI based cache analysis proposed in [9] (for single core) and [15] (for multiple cores). Chronos employs detailed micro-architectural modeling (superscalar, out-of-order pipeline and branch prediction).

For model checking purposes, we use C bounded model checker (CBMC) [70]. CBMC formally verifies ANSI-C programs through bounded model checking (BMC) [71]. For a given system/program P, BMC unwinds P to a certain depth. After unwinding, a Boolean formula is obtained that is satisfiable if and only if there exists a counter example trace. The formula is checked by a SAT procedure. If the formula is satisfiable, a counter example is produced from the output of SAT procedure. Technically, for a C program, the unwinding is achieved

by unrolling the program loops to a certain depth. For a given unwinding depth n, CBMC unwinds a loop by duplicating the code of loop body n times. Each copy is guarded by the loop entry condition and hence, covering the cases where the loop executes for less than n iterations. The main advantage of CBMC is that the tool also checks whether sufficient unwinding has been done and thereby ensures that no *longer* counterexample can exist. Technically, CBMC achieves the same by putting an "assertion" (called *unwinding assertion*) after the last copy of the unrolled loop. The assertion uses the negated loop entry condition and therefore, it ensures that the program never requires more iterations. *In summary, if no counterexample is produced by CBMC, it ensures the absence of error in the program for any execution.*

As described in the preceding, CBMC requires unwinding depth (bound) of each loop. If user does not specify any unwinding depth (loop bound), CBMC tries to determine the depth automatically. In most of our experiments, CBMC was able to determine the loop bound automatically. For the cases where CBMC failed to determine the loop bound, we passed sufficient loop bound for each loop as an input to CBMC. Recall that CBMC automatically put an "assertion" clause (called an *unwinding assertion*) after the last unwound copy of a loop. The assertion clause verifies the negated loop entry condition. Therefore, if insufficient loop bound is provided by the user, CBMC generates an unwinding assertion violation and the verification process returns a failure. Consequently, user can give a larger loop bound and rerun CBMC. However, in our experiments, we initially provided sufficient loop bounds, so that no unwinding assertion is violated. In our current implementation, CBMC is called as an external module. Therefore, for each different call of CBMC, the loop unwinding needs to be performed. Running time of our analysis can certainly improve if we can restrict the number of loop unwindings. This will require us to make use of CBMC and Chronos in a single binary executable, which could be explored in future.

Figure 5.5 gives an overall picture of our implementation framework. The figure demonstrates one refinement for each type of conflicts. Chronos employs AI based cache analysis directly on the executable. We use a utility `addr2line` which converts an instruction address to corresponding source code line number. The information generated by `addr2line` is used to generate the transformed code. The transformation of code is *entirely automatic*. Note that the sole purpose of the transformed code is to prove that certain cache conflicts in the original code are *infeasible*. Therefore, the timing effects generated by the original code is entirely independent of the additional code introduced in transformation. The transformed code contains an

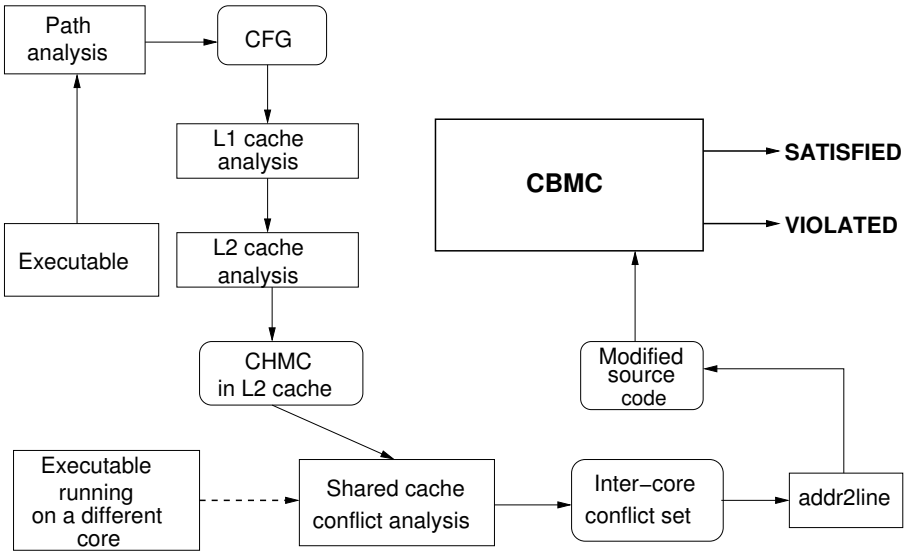


Figure 5.5: Implementation framework using CBMC

"assertion" property to be verified by CBMC. CBMC either successfully verifies the assertion property or generates a counter example. We would finally like to point out that the central contribution of this paper is an efficient composition of abstract interpretation and model checking. Therefore, even though we have used CBMC for model checking, our proposed framework (Figure 5.1) remains unchanged if we use a model checker that directly works on the executable (*e.g.* [72]). Nevertheless, there are certain advantages of using a model checker like [72]. Since [72] directly works on the executables, it can capture the effect of all compiler optimizations. Our technique can be integrated with [72] to make a more *robust* WCET analysis framework.

5.4.2 Experimental setup

We have chosen benchmarks from [2] which are generally used for timing analysis. Note that the main motivation of our work is to remove *spurious cache conflicts*, which were introduced due to the infeasible paths. Infeasible paths are often introduced when auto generating code from a high level modeling language (*e.g.* `esterel` as shown in [22]). For evaluation of our framework, therefore, we need a set of tasks which potentially exhibit many paths. Table 5.1 demonstrates a set of benchmarks having multiple paths. Let us call the set of tasks in Table 5.1 as *conflicting task set*. All the model checker (CBMC) passes are used to refine the inter-core conflicts generated by the conflicting task set. We use another set of benchmarks from [2] as shown in Table 5.2 during inter-task and inter-core conflict refinement. We call the tasks in Table 5.2 as *standard task set*. During inter-core cache conflict refinement, we refine the

conflicts generated by the conflicting task set on the standard task set. We report our experiences for each possible combinations of standard and conflicting task set.

Table 5.1: Conflicting task set

Task	Description	code size (bytes)
statemate	Automatically generated code from Real-time-Code generator STARC	52618
compress	Data compression program	13411
nsichneu	Simulate an extended petri-net	118351

Table 5.2: Standard task set

Task	Description	code size (bytes)
cnt	Counts non-negative numbers in a matrix	2880
fir	Finite impulse response filter	11965
fdct	Fast discrete cosign transform	8863
jfdctint	discrete cosign transform on 8×8 block	16028
edn	signal processing application	10563
ndes	complex embedded code	7345

We use the following terminology in presenting the experimental data:

- $WCET_{base}$: WCET before any refinement by model checker.
- $WCET_{refined}$: WCET after refinement by model checker.

WCET improvement is computed as $\frac{WCET_{base} - WCET_{refined}}{WCET_{base}} \times 100\%$.

Our framework uses the usual 5-stage pipeline (IF-ID-EX-MEM-WB) implemented by Chronos when predicting the WCET value. We fix the L1 cache miss latency as 6 cycles and L2 cache miss latency as 30 cycles for all the experiments. For the experiments which do not have an L2 cache (*e.g.* inter-task and intra-task conflict refinement), we simply take the L1 cache miss penalty as 36 cycles. All reported experiments have been performed in a 3 GHz Pentium 4 machine having 1 GB of RAM and running ubuntu 8.10 operating system. The reported total time captures the entire time taken during the analysis — including the base analysis through abstract interpretation and repeated CBMC invocation steps.

5.4.3 Evaluation

Key result Before going into the details of each experiment, let us first demonstrate the key result of this work via Figure 5.6. Figure 5.6 shows the average WCET improvement using

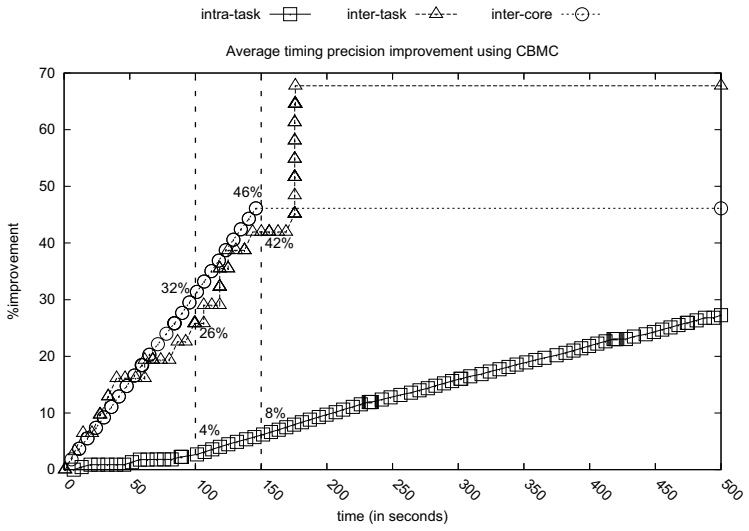


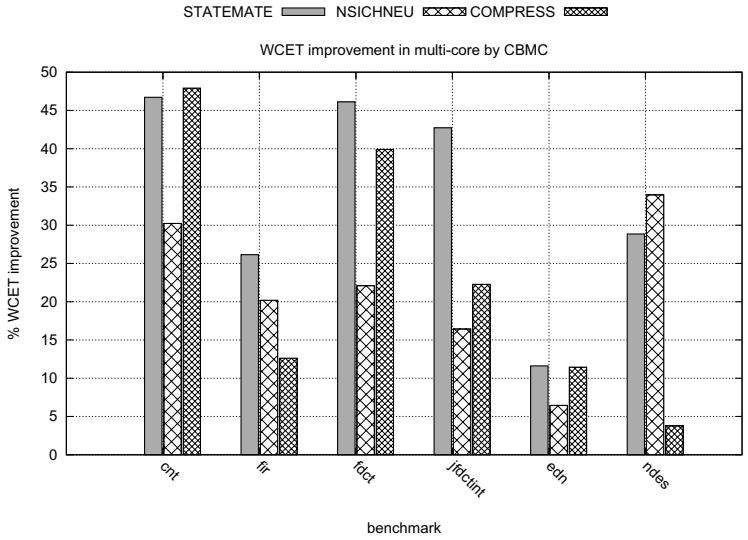
Figure 5.6: Timing precision improvement w.r.t. time using *statemate* and CBMC

CBMC. The improvement of WCET is demonstrated in case of inter-core cache conflict refinement. We observe that inter-core cache conflict refinement demonstrates an almost linear improvement in timing precision (*i.e.* improvement in WCET) with respect to time.

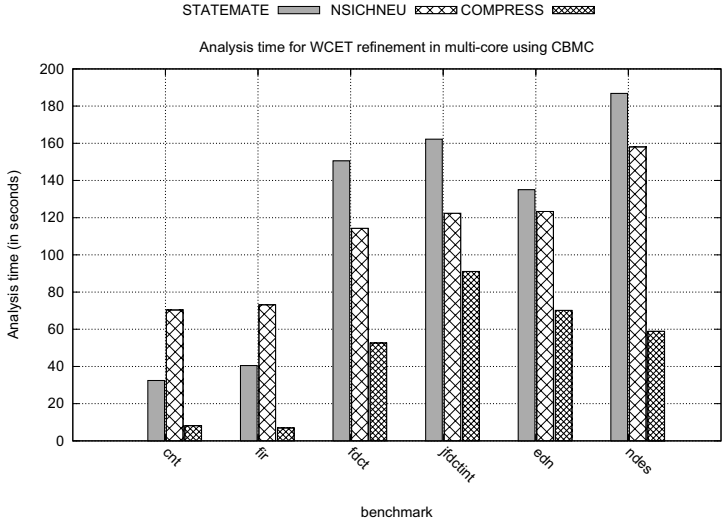
As our result is always *safe*, a provably correct WCET value can be obtained from any vertical cut along the time axis of Figure 5.6. As illustrated in Figure 5.6, consider the vertical cut at 100th second. It clearly shows that if we end the model checker (CBMC) refinement process after 100 seconds, we can obtain 32% improvement during inter-core cache conflict refinement. Nevertheless, if the model checker refinement process is allowed more time to run, we can obtain better precision in our obtained result (46% for inter-core conflict refinement after 150 seconds, as shown in Figure 5.6).

Reducing inter-core cache conflicts Finally, we present the result of inter-core cache conflict refinement in Figure 5.7(a). The analysis time recorded for each refinement is reported in Figure 5.7(b). In one core, we run a task from the standard task set (in Table 5.2) and in another core, we run a task from the conflicting task set (in Table 5.1). Reported WCET improvements represent the WCET improvements from the standard task set. For the experiments reported in Figure 5.7(a), we need the analysis of both L1 and L2 cache. We fixed the L1 cache as a direct-mapped, 256 bytes with a block size of 32 bytes. L1 cache is taken relatively small so that we are able to generate reasonable number of conflicts in the shared L2 cache. We take a 4-way associative, 8 KB shared L2 cache having a cache block size of 32 bytes.

We are able to significantly reduce the standard task WCET by refining the inter-core cache



(a)



(b)

Figure 5.7: (a) WCET improvement in multi-core using CBMC, (b) analysis time using CBMC

conflicts (maximum improvement around 50%). Similar to the inter-task cache conflict refinement, we run the refinement process until we had checked all possible and spurious inter-core cache conflicts. All our experiments using CBMC complete within four minutes.

5.5 Cache conflict refinement through symbolic execution

Motivation Our compositional analysis framework using abstract interpretation and constraint solving is inspired by the recent advances in *satisfiability modulo theory* (SMT) and *program path exploration*. In the past few years, constraint solver based path exploration has made significant progress for program functionality testing [73; 74]. In these works, different feasible

program paths are explored to find *functionality bugs*. Our work combines constraint solving with abstract interpretation to reduce the imprecision of abstract interpretation based cache analysis.

In this section, we shall extend our compositional analysis framework with a symbolic execution engine. As before, we use the abstract interpretation (AI) as a base analysis. We rule out the set of inter-core cache conflicts which are accurately analyzed by AI. Rest of the cache conflicts are iteratively refined using our code transformation framework and a symbolic execution engine.

5.5.1 KLEE symbolic execution engine

KLEE [75] is a symbolic execution engine based on LLVM [76] compiler infrastructure. KLEE uses the power of satisfiability modulo theory (SMT) and the SMT based solvers to explore different paths in a program. Such a path exploration strategy has been proved very effective in exposing some critical functionality bugs in real-world programs [74].

To better understand the workflow of KLEE, we shall use our example in Figure 5.2. Although KLEE interprets the LLVM bitcode, for the sake of simplicity, we shall convey the main idea through the source code shown in Figure 5.8.

Assume that z represents an input to the program shown in Figure 5.8. Before KLEE starts interpreting the program, a few variables of the program are marked as *symbolic*. Typically, these symbolic variables represent the input to the program. Any expression, whose value depends directly or indirectly on these symbolic variables, are treated as symbolic expressions throughout the program. For the program in Figure 5.8(a), we mark z as *symbolic*, as z is an input to the program. If the value of an expression does not depend on any of the symbolic variables, the expression value is treated as *concrete* (*i.e.* input independent). In Figure 5.8, any update on variable i and x are interpreted as *concrete* values, as the updates on i and x are not data dependent on the value of z.

At each program point, KLEE maintains a constraint store. The constraint store is a symbolic formula on the input variables which *must be satisfied to reach the same program point*. The constraint store is *the logical formula true* at the beginning of the program and is adjusted at each branch instruction. In example 5.8(b), the program hits the $i < 100$ branch instruction first. Since i is not an input and is initialized 0, only the *true* leg of the branch instruction is interpreted.

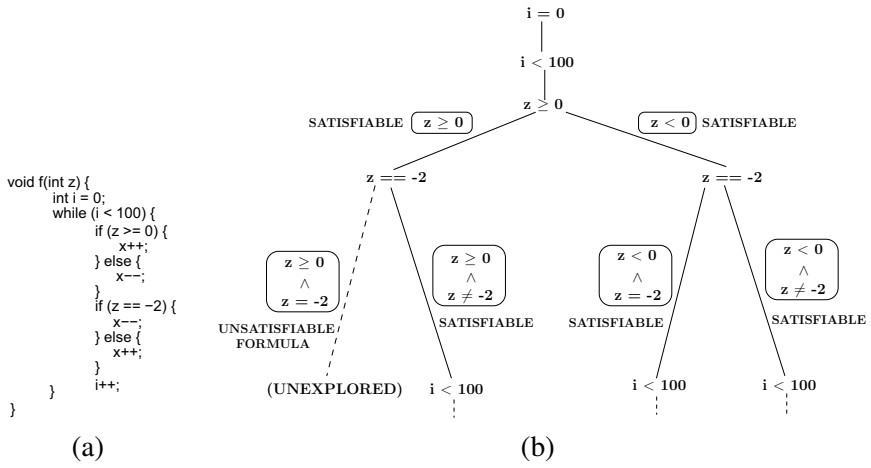


Figure 5.8: (a) Example program, (b) KLEE symbolic execution

However, consider the branch instruction $z \geq 0$, when being hit for the first time. At this point, the constraint store is *the logical formula true*. This branch condition is sent as a query to the SMT solver to decide the condition outcome (*i.e.* true or false). The SMT solver consults the constraint store to decide the outcome of the branch condition. Since the constraint store is *the logical formula true*, the outcome of $z \geq 0$ could be both *true or false* depending on the value of input z. Therefore, KLEE forks two different execution states for each leg of the branch instruction. The constraint store at the true leg is updated as $z \geq 0$ and the same at the false leg is updated as $z < 0$. The content of the constraint store is shown beside the control flow edges.

Now consider the branch instruction $z == -2$ with constraint store $z > 0$. The SMT solver checks the satisfiability of the formula $z \geq 0 \wedge z = -2$, which is clearly *unsatisfiable*. The unsatisfiability of such formula can be checked very fast by an SMT solver with the theory of linear integer arithmetic. Therefore, KLEE does not create any execution state which corresponds to the unsatisfiable constraint store $z \geq 0 \wedge z = -2$. Eventually, three different execution states are created (as shown in Figure 5.8(b)) with their respective constraint stores as follows:

- $z \geq 0 \wedge z \neq -2$,
- $z < 0 \wedge z = -2$, and
- $z < 0 \wedge z \neq -2$

The symbolic execution by KLEE is terminated when it finishes interpreting all the instructions in all the three execution states (as shown in the preceding).

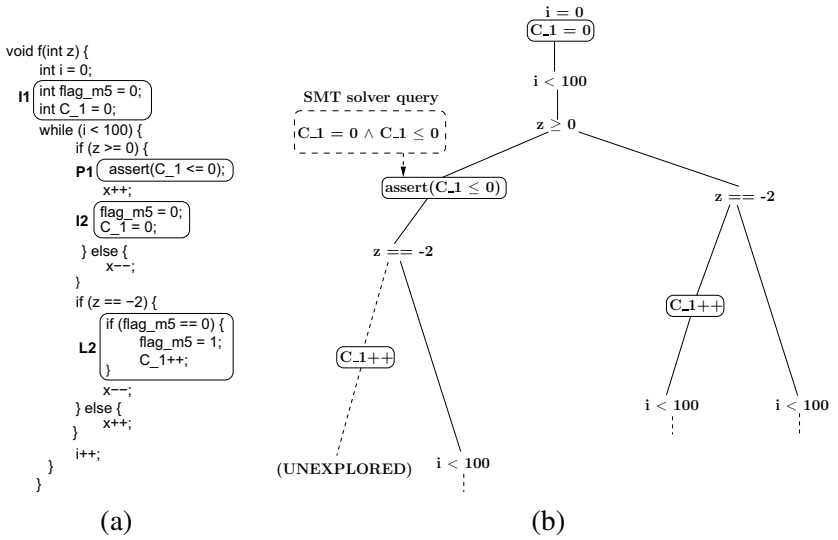


Figure 5.9: (a) Transformed code for checking cache conflict, (b) checking the assertion during KLEE symbolic execution

5.5.2 Cache conflict refinement

KLEE has successfully been applied to discover many critical functionality bugs. At a high level, our code transformation framework can be viewed as reducing the problem of cache timing checking to functionality checking. Recall that our code transformation framework contains an assertion property $\mathcal{A}$ to check whether certain cache conflicts in the program are *spurious*. This assertion property can be checked for validity using KLEE. If any execution of $\mathcal{A}$ leads to a violation of the property captured by $\mathcal{A}$, the entire symbolic execution by KLEE is *aborted*. Such an abnormal termination of the program captures the fact that certain cache conflicts (captured by $\mathcal{A}$) can be realized for some execution of the program and therefore, such cache conflicts are not *spurious*. On the other hand, if the execution of KLEE is not aborted, we can prove that our introduced assertion holds over all possible executions of the program. Consequently, the cache conflict captured by the assertion property is *spurious*.

We shall demonstrate the refinement process through the example in Figure 5.9. Figure 5.9(b) shows that only one execution state (among all three) can execute the assertion property involving the variable C_1. Since KLEE interprets the program, at each program point it holds the value of all the registers and memory locations. At the assertion location, KLEE checks whether the currently stored values satisfy the assertion. Since C_1 has a value of zero, a formula of the form $C\_1 = 0 \wedge C\_1 \leq 0$ is sent to the SMT solver as a query. If the SMT solver returns a *satisfiable* formula, we can conclude that the assertion property holds for the

corresponding execution. For the example shown, all the executions of the assertion property send the same formula (*i.e.* $C\_1 = 0 \wedge C\_1 \leq 0$) to the SMT solver. Therefore, KLEE execution is never aborted for the example and we can conclude that $m1$ and $m5$ cannot create conflicts in the cache for *any execution*.

It is important to note that the above checking procedure is entirely different from CBMC. In CBMC, the checking of an assertion property is captured by a single SAT formula. The SAT formula takes care of all the different program paths that may reach the assertion. Therefore, in general, the SAT formula created by CBMC is very large. On the other hand, KLEE does not check the assertion by a single formula. KLEE checks the assertion property while interpreting the program. Therefore, each time the assertion is interpreted, an SMT solver is asked to check the satisfiability of the assertion. The symbolic and concrete values at the assertion location are used to validate the assertion property. Note that each interpretation of the assertion captures a single program path and therefore, the formula checked by the SMT solver is usually much simpler than the single SAT formula generated by CBMC. Nevertheless, an SMT solver is called many times to check the assertion property, whereas CBMC calls a SAT solver only once.

Finally, for a violation of the assertion property, the KLEE symbolic execution can be aborted as soon as a violation is reached. As a result, a violation of the assertion is likely to be checked much more quickly than the *validity* of the same assertion. On the other hand, since CBMC creates a single SAT formula capturing all the program paths, it has to wait till the formula is generated and checked for satisfiability by the SAT solver. Therefore, the time taken by CBMC for the violation (or validity respectively) of an assertion largely depends on the performance of the SAT solver to check the satisfiability (or unsatisfiability respectively) of a formula.

5.6 Implementation and evaluation using KLEE

5.6.1 Implementation

Figure 5.10 shows our implementation framework using KLEE. The basic structure of the implementation is same as in Figure 5.5. The modifications made to use KLEE have been highlighted in Figure 5.10. KLEE is a symbolic execution engine based on the LLVM bitcode format. Therefore, our transformation is made at the level of LLVM bitcode. KLEE allows to specify assertions, which are checked during the symbolic execution using [77] constraint solver.

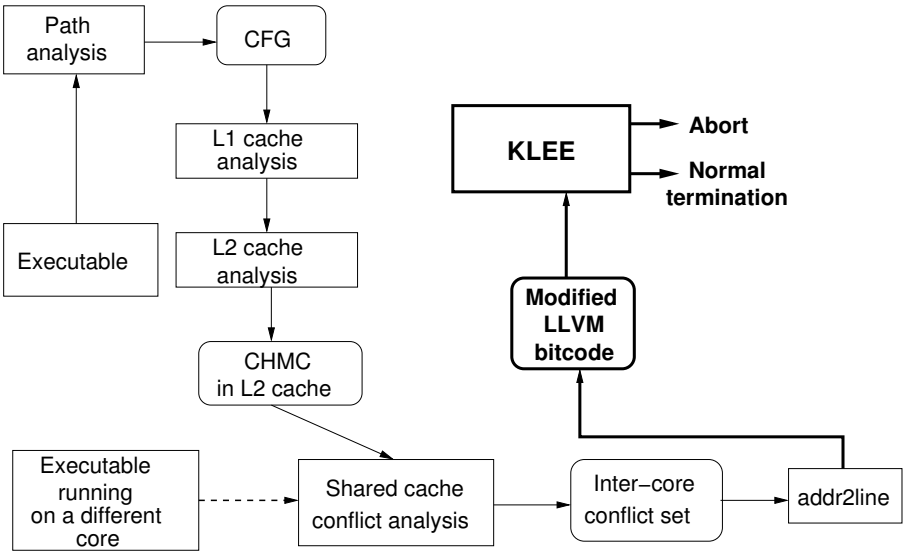


Figure 5.10: Implementation framework using KLEE

Originally, KLEE ignores the assertions with a warning and continues symbolic execution. We modify the source code of KLEE to terminate the symbolic execution as soon as it reaches the violation of some assertion property. Note that our sole purpose is to check the assertions introduced in the modified code, and therefore, we do not need to continue execution if the assertion is violated in some execution state. As a result, KLEE can usually check the violation of an assertion property much faster than CBMC.

5.6.2 Evaluation

In this section, we shall evaluate our compositional analysis framework using symbolic execution engine (*i.e.* the implementation framework shown in Figure 5.10). We shall compare the results obtained using symbolic execution (*i.e.* using KLEE) with the results obtained using model checking (*i.e.* using CBMC). To make a *fair* comparison, we use the same experimental setup of Section 5.4.2. Therefore, for each of the experiments reported in the following, we use the exactly same micro-architectural configuration and application setting used in Section 5.4.3 (*i.e.* during the evaluation of our framework using model checking). We shall also compare the overall analysis time required for our framework using model checking and symbolic execution.

Key result Figure 5.11 shows the average WCET improvement using KLEE. The improvement of WCET is demonstrated in case inter-core cache conflict refinement.

We observe from Figure 5.11 that the analysis time using KLEE is much smaller compared

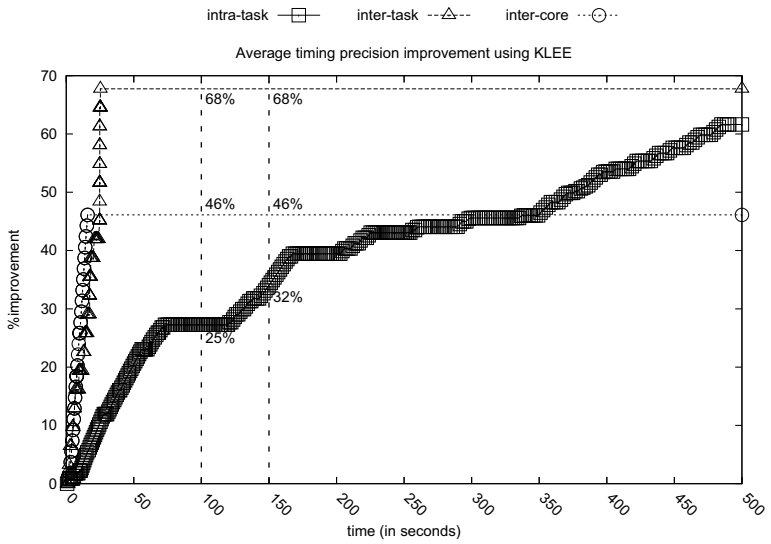


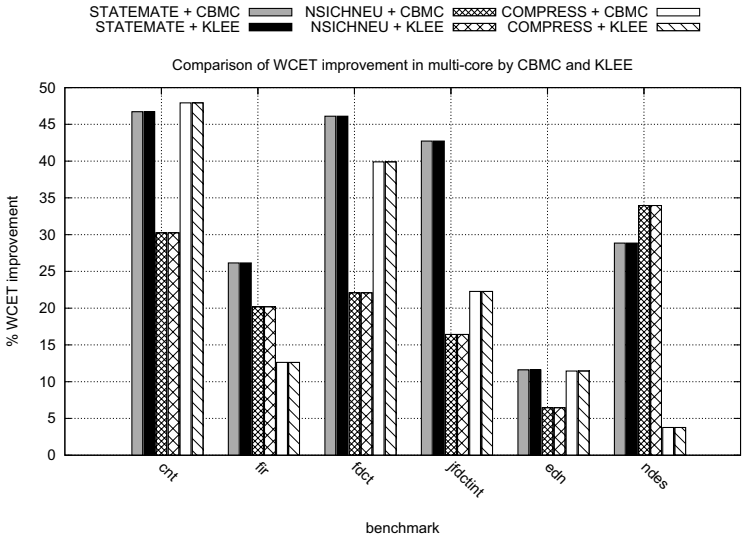
Figure 5.11: Timing precision improvement w.r.t. time using `statemate` and KLEE

to the time taken by CBMC (refer to Figure 5.6). Similar to model checker refinement phase, our analysis result is always *safe* during the refinement through symbolic execution. However, as the symbolic execution through KLEE is much faster than model checking, we can obtain a provably correct, yet precise WCET value using KLEE quicker than using CBMC. As illustrated in Figure 5.11, consider the cut at 15th second. By 15 seconds, KLEE is able to check all possible inter-core cache conflicts.

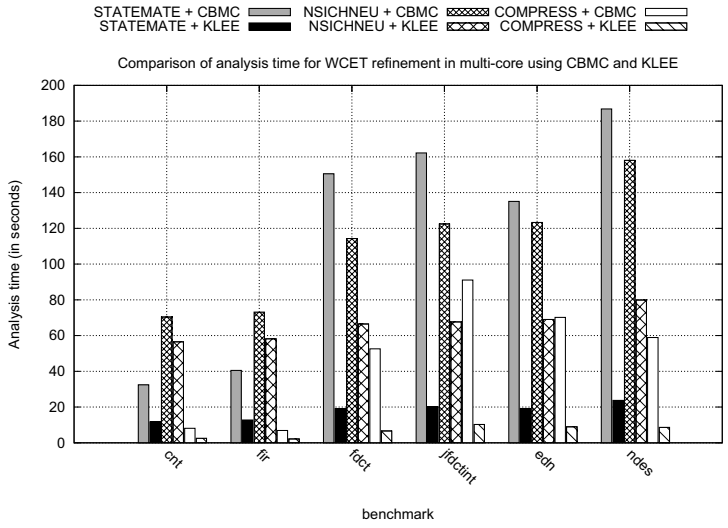
Together from Figure 5.6 and Figure 5.11 we can conclude that the improvement in precision using symbolic execution can be obtained faster than model checking. This is evidenced by the vertical cuts along the time axes in Figure 5.6 and Figure 5.11.

Reducing inter-core cache conflicts Figures 5.12(a) shows the precision gain obtained using KLEE. KLEE and CBMC produce the exactly same precision gain in WCET. For all the benchmarks, KLEE and CBMC are able to refine the same number of inter-core cache conflicts, thereby reducing the WCET by the exactly same amount. This result is evidenced by Figure 5.12(a).

Figure 5.12(b) compares the analysis time overhead using CBMC and KLEE. For `nsichneu` and `compress`, KLEE generates the same results at least twice faster than CBMC. On the other hand, for `statemate`, usage of KLEE leads to a significant improvement in the refinement process – with as much as 900% for a few benchmarks (Figure 5.12(b)). The maximum time taken by KLEE for any of the benchmarks is 80 seconds.



(a)



(b)

Figure 5.12: (a) Comparison of multi core WCET improvement using CBMC and KLEE, (b) comparison of analysis time using CBMC and KLEE

5.6.3 Discussion

We have evaluated our framework using two different forms of iterative refinements – model checking and symbolic execution. For model checking, we have used CBMC and for symbolic execution, we have used KLEE. In our experiments, CBMC was unable to infer some of the loop bounds in a program automatically. In particular for `statemate`, we provided sufficient loop bounds, so that no unwinding assertion is violated (recall that an unwinding assertion is violated when the user given loop bound may under-approximate the number of times the loop body can be executed). On the other hand, since KLEE performs a symbolic execution of the program, it was able to automatically detect the termination of all the loops and no manual intervention was required during the experiments using KLEE. Our evaluation shows that both the symbolic execution and model checking improve the analysis precision by exactly same amount. However, a symbolic execution guided refinement process is much *faster* than the refinement process based on model checking. This time efficiency of symbolic execution has been made possible by the recent advances in SMT technologies. KLEE uses the fast SMT solver STP for constraint solving, hence improving the refinement process of our framework significantly. Moreover, if any execution of an assertion property leads to a violation, the entire symbolic execution by KLEE can be terminated. This in turn makes the violation check of an assertion much faster than CBMC.

5.7 Chapter summary

In this chapter, we have proposed two compositional WCET analysis frameworks, one of which combines abstract interpretation with model checking and the second one combines abstract interpretation with constraint solving, both for shared cache modeling. Our framework does not affect the flexibility of abstract interpretation based cache analysis and it can be composed with the analysis of different other micro-architectural features (*e.g.* pipeline). Moreover, our model checker or symbolic execution guided refinement process is always *safe*. Therefore, the refinement process can be terminated at any point if the time budget is violated. Experimental results show that we can obtain significant improvement in cache analysis for multi-cores using both of our compositional analysis frameworks.

Chapter 6

Modeling Shared Cache and Bus for Timing Analysis

In Chapter 5, we have modeled the timing effects of shared cache. In this Chapter, we shall extend our framework to analyze the timing effects of another primary shared resource in multi-core – namely the shared bus. It is very common that the shared last level cache or the external memory is accessed through a shared bus. Shared cache and shared bus introduce unpredictable execution time behaviour of a program due to the conflicts arising from different cores. The conflicts arising in the shared cache and the shared bus are not independent. Therefore, it is crucial to model the timing effects of both the shared cache and the shared bus and their interactions for current generation multi-core architectures. *To the best of our knowledge, ours is the first work to model the timing effects of both the shared cache and the shared bus*.

6.1 System and Architectural Model

Our system architecture is representative of the current generation of commercial multi-core platforms (Figure 6.1). Each core on chip has one or more levels of private caches and the last level of private caches from all the cores are connected to a large shared cache through a shared bus. For example, ARM Cortex-A9 MPCore [78] and Intel core 2 (code named Penryn) [79] have only private L1 caches that are connected via a bus to the shared L2 cache as shown in Figure 6.1 (Architecture A). The advantage of a shared cache is that the cache space can be dynamically and transparently allocated to the different cores based on their memory requirement. Next-generation multi-cores are likely to introduce more levels of private caches before

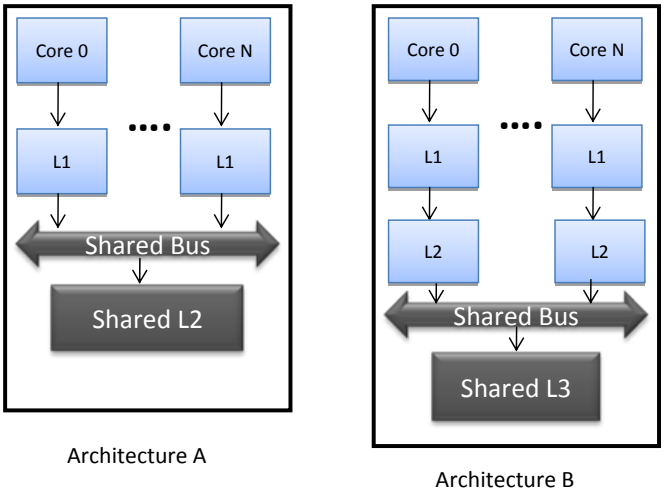


Figure 6.1: Multi-core cache memory hierarchy.

hitting the shared resources. For example, Intel Xeon [80] has private L1 and L2 caches; the L2 caches are connected to a large shared 16MB L3 cache through a bus as shown in Figure 6.1 Architecture B.

In this work, we will assume, without loss of generality, the first architecture in Figure 6.1 (Architecture A) to develop WCET analysis of shared resources in multi-core platforms. Extending our work to multiple levels of private cache hierarchy is simply a matter of employing the same propagation principle that we employ from L1 to L2 cache.

We focus here only on the instruction memory. We assume that the data memory references do not interfere with the L1 and L2 instruction caches modeled by us (they could be serviced from a separate data cache that we do not model). We do not allow self-modifying code and hence do not need to model cache coherence. For each program, all shared library code used in it are copied into its private code section. Hence, there is no code sharing among different programs running in different cores. We consider Least Recently Used (LRU) cache replacement policy for set-associative caches. Also, we consider architectures without timing anomalies caused by interactions between caches and other architecture features. The L2 cache block size is assumed to be larger than or equal to the L1 block size. This is usually the case in real architectures to exploit higher spatial locality through a second level cache. Finally, we are analyzing non-inclusive multi-level caches [65].

The shared communication infrastructure in our architecture is the bus. It is used for accessing the instructions and data from the shared L2 cache (in case of L1 cache miss) by the different cores. However, as we are not modeling the data caches, we assume fully separated

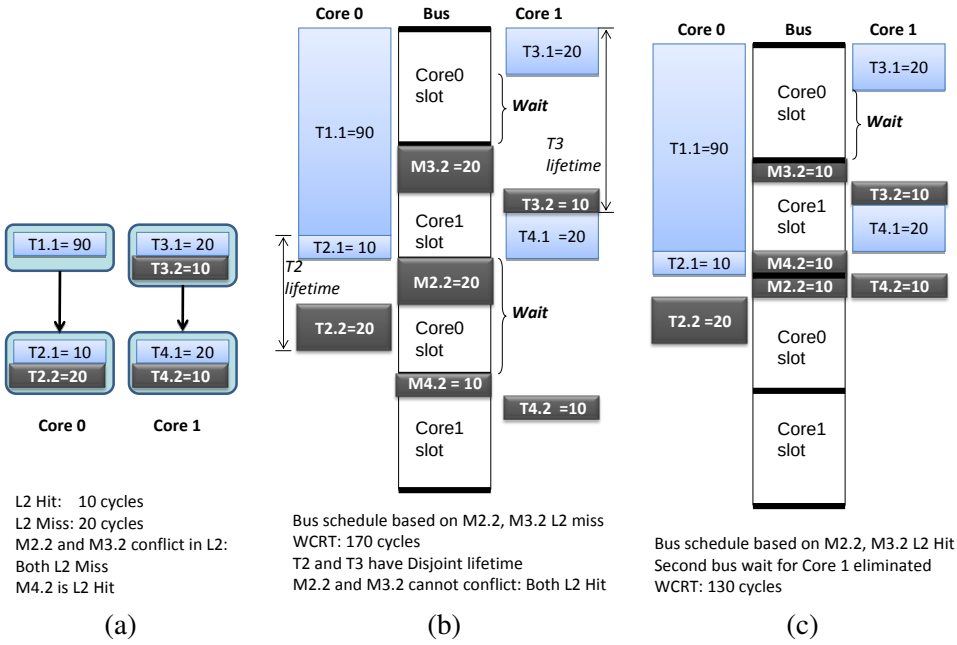


Figure 6.2: Example to show dependency between cache and bus analysis.

buses and memories for both code and data. Therefore, we ignore bus traffic arising from data memory accesses and this includes interprocess communication through shared memory. We assume TDMA-based static bus scheduling policy where a fixed length bus slot is allocated to each core in a round-robin fashion.

During WCET analysis, we assume all loop bounds are known through user annotation or simulation. We also assume all paths in a program are feasible and all loops in a program are reducible (i.e., all loops have a single entry and single exit).

We model the application as a set of task graphs. Each task graph is a directed acyclic graph consisting of a number of tasks. Let $\{T_0, \ldots, T_{N-1}\}$ be the set of N tasks corresponding to all the task graphs. A directed edge between two tasks T_i and T_j in a task graph denotes that task T_j can start execution only after task T_i completes execution. Our objective is to estimate the worst-case response time (WCRT) of the overall application.

6.2 Overview

Our WCRT analysis framework in the presence of shared cache and bus in multi-core platforms appears in Figure 6.3. L1 cache analysis proceeds independently for each core. The memory accesses that are guaranteed to be L1 cache hits are eliminated from further consideration at this point. The remaining memory accesses (guaranteed / probable L1 misses) can be transmitted

via the bus and are considered for shared cache and bus analysis.

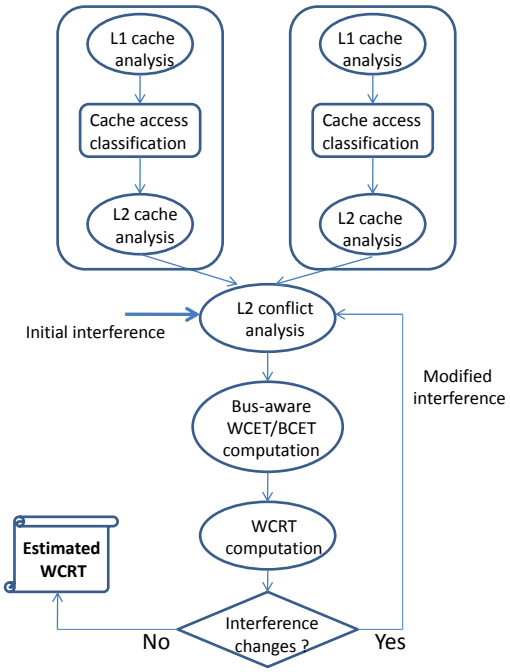


Figure 6.3: Our analysis framework

Clearly, the bus analysis requires the time at which the L1 cache misses appear on the bus. However, the bus access time of an L1 cache miss is affected by the execution time of the preceding memory accesses in the same core. This in turn is determined by the shared L2 hit/miss categorization of the preceding memory accesses. On the other hand, the shared L2 cache conflict analysis determines the memory blocks that may get evicted by memory blocks from other core. Whether a memory block $M1$ belonging to task $T1$ can be evicted from the shared cache by a memory block $M2$ from task $T2$ depends on whether the lifetime of the two tasks can overlap or not. The task lifetime, in turn, is determined by the shared bus analysis results.

This circular dependency between the bus and cache analysis requires us to develop an iterative analysis framework as shown in Figure 6.3. In the first iteration, we perform shared L2 cache analysis assuming that a task on one core can conflict with all the tasks in other cores. Based on this pessimistic L2 cache analysis results, we estimate the shared bus access time and hence the WCET of the different tasks. These numbers are fed to the WCRT analysis component that estimates the worst-case response time of the complete application by taking into account the dependencies among the tasks. A by-product of the WCRT analysis framework is the lifetime of each task. These lifetime estimates are used to eliminate interference among

tasks with disjoint lifetimes. If the interference pattern has changed (i.e., we have managed to eliminate some interferences), the shared L2 cache analysis has to be repeated. We can formally prove that our analysis monotonically reduces the task interferences across iterations, and hence is guaranteed to terminate.

Illustrative Example We now show the working of our analysis using the example in Figure 6.2(a). We assume a 2-core system where the task graph containing tasks $T1$ and $T2$ are running on core 0 and task graph containing tasks $T3$ and $T4$ are running on core 1. For simplicity of exposition, we shall assume in this example that *best case* and *worst case* execution times of any task are same. $T1.1, T2.1, \ldots, T4.2$ represent the memory blocks within the tasks. Each memory block is annotated with its computation cost. Only the memory blocks marked in black are the ones with guaranteed or possible L1 cache miss as determined by per-core L1 cache analysis. We perform an initial L2 cache analysis for each core individually that ignores conflicts from other cores. This per-core L2 cache analysis determines all the memory blocks ($T2.2, T3.2$, and $T4.2$) as guaranteed L2 cache hits. Let us also assume that L2 cache hit latency is 10 cycles, whereas L2 cache miss latency is 20 cycles. Further, the round-robin TDMA bus scheduler assigns a 50 cycle bus slot to each core and the first bus slot goes to core 0. In this example, to demonstrate the dependency between shared cache and bus analysis, we ignore any *cold cache misses*. However, our analysis does not rely on that assumption and it accurately models the additional cycles due to cache misses if some memory blocks have to be loaded into the cache for the very first time.

Now we proceed to shared L2 cache analysis. At this point, we have no information about task lifetimes. So we assume any task on core 0 can conflict with all the other tasks on core 1 and vice versa. Memory block $T2.2$ and $T3.2$ map to the same L2 cache block and therefore they conflict with each other. So we have to conservatively assume that both of them will be L2 cache misses in the worst case, whereas $T4.2$ remains as L2 cache hit because it does not conflict with any memory block from core 0. Note that, even though any task on core 0 can conflict with all the other tasks on core 1 and vice versa, memory block $T4.2$ may not conflict with $T2.2$ since it maps to a different cache block in shared L2 cache.

After shared L2 cache analysis, we proceed to shared bus analysis. The result of the analysis can be visualized in Figure 6.2(b). In Figure 6.2, a memory transaction corresponding to the L1 cache miss of memory block $Px.y$ is denoted by $Mx.y$. Notice that all L2 cache accesses

(whether hit or miss) are transmitted on the shared bus in our architecture. An L2 cache access from core i has to wait for core i to get access to the bus. The L1 cache miss $M2.2$ in core 0 occurs at time 100. From the bus schedule, we can observe that the slot beginning at time 100 belongs to core 0. Thus $M2.2$ does not encounter any additional waiting time to acquire the shared bus and is completed by time 120. Thus, $T2$ finishes at time 140. However, the L2 cache miss $M3.2$ in core 1 happens at time 20 and the bus slot from time 0 to time 50 is alloted to core 0. Hence, $M3.2$ encounters an additional 30 cycles waiting time to acquire the bus and eventually the memory transaction corresponding to $M3.2$ completes at time 70. This makes task $T3$ to finish at time 80. Similarly, the L2 cache hit $M4.2$ in core 1 occurs at time 100 and the bus slot from time 100 to time 150 is alloted to core 0. Thus $M4.2$ encounters an additional 50 cycles waiting time and eventually the task graph running on core 1 is completed at time 170. Hence, the WCRT of the application according to this schedule is 170 cycles.

However, as a by-product of the WCRT analysis, we note that task $T2$ and $T3$ have disjoint lifetimes. So memory blocks $T2.2$ and $T3.2$ cannot conflict with each other in the shared L2 cache and they remain as L2 cache hits as determined by per-core L2 cache analysis. As L2 cache hits have shorter latency, the bus analysis needs to be re-done. The revised schedule is shown in Figure 6.2(c). Task graph running on core 0 finishes at time 130 because $M2.2$ is now a L2 cache hit. Due to the earlier completion of $M3.2$ (because of L2 hit), L2 cache hit $M4.2$ occurs at time 90. Since L2 cache hit latency is 10 cycles, $M4.2$ can be serviced in the remaining bus slot belonging to core 1 (i.e., the bus slot from time 90 to time 100) and therefore making $T4$ finish by time 110. Hence, this new analysis results in much tighter WCRT estimate as the second wait time for the bus in core 1 is now eliminated. The WCRT at this point changes to 130 cycles. This example illustrates how an iterative shared cache and bus analysis can obtain tight WCRT estimates for embedded real-time applications.

6.3 Bus aware WCET analysis

We now present a bus-aware WCET analysis of programs. Note that L1 cache misses are transmitted via the bus to access the shared L2 cache (Fig. 6.1, Architecture A).

Classical WCET analysis can compute the WCET of a program by taking into account only the number of worst case cache misses. The exact time-stamp of the cache misses (the time at which the cache misses occur) are not required for WCET computation. In presence of a shared

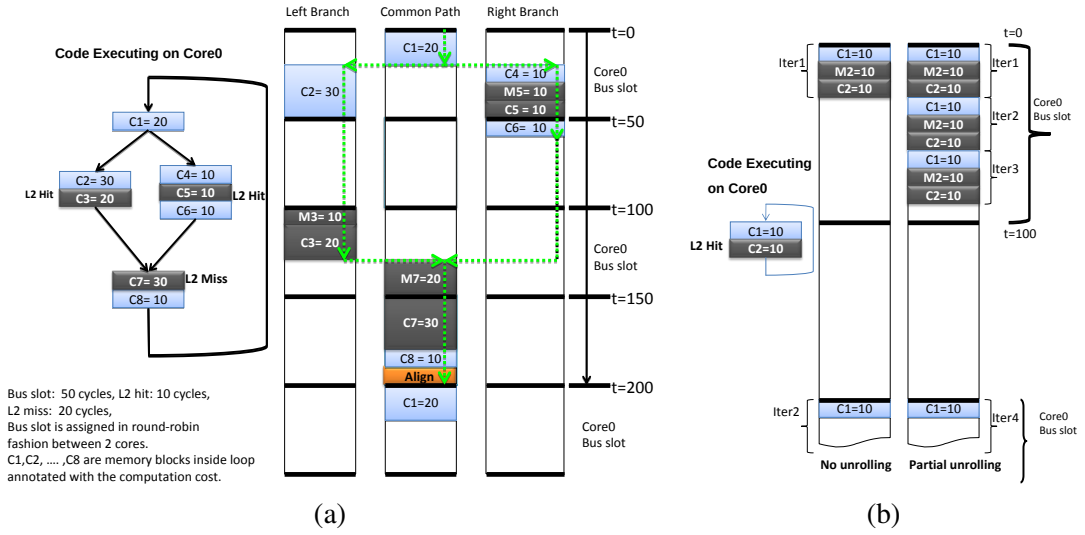


Figure 6.4: (a) An example of loop analysis (b) Limited loop unrolling for loop iterations with low cost.

bus, a cache miss encounters variable amount of delay due to the waiting time elapsed to acquire the bus-slot for the corresponding core. One naive approach is to always consider the maximum possible waiting time for each memory reference that may potentially access the shared bus. In that case, effect of shared bus in WCET analysis can be ignored at the cost of obtaining highly over-estimated WCET value. Our analysis effectively bounds the over-estimation in WCET analysis, while keeping the analysis time-efficient.

Formally, the round-robin TDMA bus schedule is represented by the following recurrence relation:

$$CS_k^{(i+1)} = CS_k^{(i)} + B; \;\; CS_k^{(0)} = A_k \tag{6.1}$$

where $CS_k^{(i)}$ is the starting time of the bus schedule assigned to k-th core in i-th round, $B = J \times s_l$, J being the total number of cores, s_l is the slot length assigned to each core and A_k is the starting time of the very first slot in the bus schedule assigned to k-th core.

At first we discuss the WCET computation of a single loop (no nesting) and later we extend it to a full program. Analysis of loop is depicted by an example in Figure 6.4(a). The bus slot is 50 cycles. Let us also assume that L2 cache hit latency is 10 cycles, whereas L2 cache miss latency is 20 cycles. Only the memory blocks marked in black denote L1 cache misses and hence will be transmitted via the bus. The loop starts at 0 time. Following this assumption, L1 cache miss M3 occurs at time 50. Since the next bus slot for Core0 starts only at time 100, this L2 cache access is delayed till time 100. Thus total time encountered for M3 access becomes

60 cycles — 50 cycles to wait for the bus and 10 cycles to get the instruction from L2 cache. On the other hand, L1 cache miss M5 starts at time 30, when the bus is still available to Core0. As a result, M5 does not suffer any delay to access the bus. Worst case starting time of the loop sink node is at time 130. Once again, due to the availability of the bus, L2 cache miss M7 can be served immediately. Finally the computation of loop sink node ends at time 190. Since we always assume a loop iteration starts from the beginning of a bus slot of Core0, an alignment cost of 10 cycles is added to the total cost of one iteration. Assuming loop bound to be 5, overall WCET of the loop becomes $(5 * (190 + 10) + 100) = 1100$ cycles (additional 100 cycles were added for aligning the first iteration of the loop, since the time between the beginning of any two consecutive bus slots allotted to the same core is 100 cycles). Note that, an L1 cache miss, occurred earlier than the time predicted in the worst-case, is served by an earlier bus slot (than the bus slot predicted in the worst-case analysis). This accounts for the safety of our method.

Formally, WCET computation of a loop is described in Algorithm 3. $start_{b_i}$ and $finish_{b_i}$ keep track of the worst case starting and finishing time of basic block b_i respectively. $cost$ stores the worst case cost of basic block b_i while b_i is being processed. $finish_{b_i}$ is computed by adding the value of $cost$ to $start_{b_i}$ (line 30). Header node of the loop always starts from time 0 (line 5). Worst case starting time of any basic block (other than the header node) is the maximum of all of its predecessors' finishing time (line 9). $lbus_{b_i}$ is the beginning time of the latest bus slot acquired by the core while basic block b_i is processed; this information is propagated to all successor basic blocks (line 10). For an L1 cache miss, function *Wait* computes the worst case additional delay for accessing the shared bus (line 18).

$$Wait(\Delta) = \begin{cases} 0, \text{ if } (\lfloor \frac{\Delta}{B} \rfloor \times B + s_l - LAT) >= \Delta; \\ \\ (\lfloor \frac{\Delta}{B} \rfloor + 1) \times B - \Delta, \text{ otherwise.} \end{cases}$$

Here Δ is the difference between the current time and $lbus_{b_i}$. s_l is the bus slot length assigned to each core. LAT is equal to the fixed L2 cache hit latency in case of a L2 cache hit and main memory latency in case of a L2 cache miss. The term $\lfloor \frac{\Delta}{B} \rfloor$ represent the number of *full* bus schedules (whose length is equal to B) expired in time Δ. Therefore, $\lfloor \frac{\Delta}{B} \rfloor \times B$ represents the starting time of the *latest* bus slot assigned to the core *relative* to $lbus_{b_i}$. Relative to $lbus_{b_i}$, end time of this *latest* slot is at time $\lfloor \frac{\Delta}{B} \rfloor \times B + s_l$. On the other hand, end time of the current L1

Algorithm 3 WCET computation of a loop lp; B is the interval between two consecutive bus slots assigned to a core

1. $cost_{iter}$:= 0;
2. **for** (all blocks b_i of loop lp in topological order) **do**
3. $cost$:= 0;
4. **if** (b_i is the header node of loop lp) **then**
5. $start_{b_i}$:= 0; /* assume loop header node starts at time 0 */
6. $lbus_{b_i}$:= 0; /* assume first bus slot starts at time 0 */
7. **else**
8. find the predecessor p_{max} of b_i having maximum finish time ($finish_{p_{max}}$);
9. $start_{b_i} := finish_{p_{max}}$;
10. $lbus_{b_i} := lbus_{p_{max}}$;
11. **end if**
12. $inst$:= first instruction in basic block b_i;
13. **repeat**
14. **if** ($inst$ is an L1 cache hit) **then**
15. $cost$:= $cost$ + $L1_{lat}$; /* $L1_{lat}$: L1 cache hit latency */
16. **else**
17. $\Delta := (start_{b_i} + cost) - lbus_{b_i}$;
18. **if** (Wait(Δ) > 0) **then**
19. $lbus_{b_i}$:= $start_{b_i}$ + $cost$ + Wait(Δ);
20. **end if**
21. $cost$:= $cost$ + Wait(Δ) + LAT;
22. **end if**
23. $inst$:= next instruction in basic block b_i;
24. **until** (all instructions in basic block b_i finish)
25. **if** (b_i is the sink node of loop lp) **then**
26. $\Delta := (start_{b_i} + cost) - lbus_{b_i}$;
27. $cost$:= $cost$ + AlignCost(Δ);
28. $cost_{iter} := (start_{b_i} + cost)$;
29. **end if**
30. $finish_{b_i} := start_{b_i} + cost$; /* finish time of b_i */
31. **end for**
32. **return** $cost_{iter} \times N + B$;

cache miss is $\Delta + LAT$ relative to $lbus_{b_i}$. To complete the current L1 cache miss in the *latest* bus slot, it must be the case that $\lfloor \frac{\Delta}{B} \rfloor \times B + s_l \geq \Delta + LAT$, which is precisely the first condition of $Wait$ function. If the L1 cache miss at current time cannot be served in the *latest* bus slot, it is delayed till the next bus slot. Clearly, the next bus slot starts at time $(\lfloor \frac{\Delta}{B} \rfloor + 1) \times B$ relative to $lbus_{b_i}$. Thus $(\lfloor \frac{\Delta}{B} \rfloor + 1) \times B - \Delta$ precisely represents the waiting time to acquire this next bus slot. In case a new bus slot is acquired (the second case in $Wait(\Delta)$ function), the value of $lbus_{b_i}$ is updated (line 19). After computing the worst case cost of one iteration of the loop, the additional cost to align the next iteration to the starting of a bus slot is added to the WCET (by the *AlignCost* function) (line 27). *AlignCost* function is similar to the *Wait* function and is described as follows.

$$AlignCost(\Delta) = \begin{cases} 0, \text{ if } (\Delta \ mod \ B) = 0; \\ \\ (\lfloor \frac{\Delta}{B} \rfloor + 1) \times B - \Delta, \text{ otherwise.} \end{cases}$$

Thus, if Δ is already aligned with the beginning of a bus slot alloted to the core, alignment cost is 0. Otherwise, alignment cost is equal to shift the timeline to the beginning of the *nearest* bus slot alloted to the core. By adding $AlignCost(\Delta)$ we get $cost_{iter}$, the worst case cost of one loop iteration. Since we do not know the exact starting time of the loop, for the very first iteration, maximum alignment cost needs to be added (which is equal to B). Hence, the WCET of the loop is computed as $cost_{iter} \times N + B$, where N is the loop bound.

There is a special case when the worst case cost of one loop iteration is much smaller than the bus slot length. In that case, due to the alignment to the beginning of a bus slot after one iteration, overestimation in WCET may increase significantly. We always partially unroll such loops so that worst case cost of a single iteration of the unrolled loop exceeds one single bus slot. This situation is illustrated in Figure 6.4(b). The loop is unrolled three times as L1 cache misses (M2) from three consecutive iterations can be serviced in a single bus slot.

Extension to full program So far, we have only discussed the WCET computation of a single loop. To extend our analysis to whole programs, we transform the program's control flow graph by converting each innermost loop to a single "basic block". The cost of each innermost loop is given by the pre-computed WCET. Using the innermost loop's WCET, we get the WCET of loops at the next level of nesting. In this way, we can get WCETs of all the outermost loops in a program. The program can now be viewed as a DAG with all outermost loops converted to single basic blocks. Algorithm 3 can again be used to compute the WCET of the program with zero alignment cost. For programs containing procedure calls, the extension is straightforward. For each call instruction, the cost of the callee can be computed as mentioned above and will be added to the total cost of the corresponding basic block. Our analysis is context sensitive, i.e., procedure calls at different call sites are analyzed separately. In our actual implementation, the cache analysis module also handles different contexts of a loop (i.e., *Virtual Inlining and Virtual Unrolling* (VIVU) approach [9]) and thus our shared bus analysis indeed can model different contexts of a loop. However, for simplicity of discussion, we describe only about the WCET analysis of a loop in a single context.

6.3.1 WCRT Estimation

In order to compute the WCRT of a task graph, we need to know the time interval of each task. The task ordering is imposed by the partial ordering given in the corresponding task graph. We use four variables *EarliestReady*(t), *LatestReady*(t), *EarliestFinish*(t), and *LatestFinish*(t) to represent the execution time information of a task t. For any task t, the earliest (latest) time when all of t's predecessors in the task graph have completed execution, is represented by *EarliestReady*(t) (*LatestReady*(t)). Similarly, the earliest (latest) time when task t finishes execution, is represented by *EarliestFinish*(t) (*LatestFinish*(t)). Given a task t, its execution interval is *EarliestReady*(t) to *LatestFinish*(t).

We consider a non-preemptive system. Let us assume, *WCET(t)* and *BCET(t)* denote the Worst-case Execution Time and Best-case Execution time of task t. For BCET computation, all NC classified instructions in L1 cache are considered to be L1 cache hit and all instructions that are AM classified in L1 cache and NC classified in shared L2 cache are considered to be shared L2 cache hit. BCET of all the tasks are computed after the shared L2 cache analysis. A task t can be ready only after all its predecessors *Pred(t)* in the task graph finish execution. So the following two equations hold:

$$EarliestFinish(t) = EarliestReady(t) + BCET(t) \tag{6.2}$$

$$EarliestReady(t) = \max_{u \in Pred(t)} EarliestFinish(t) \tag{6.3}$$

For a task t without any predecessor *EarliestReady(t)=0*. However, latest finish time of tasks is not only affected by its predecessors but also by the set of tasks running on the same core whose execution interval may overlap (called *peers*) [15]. Let us call the set of tasks overlapping with t, and running on the same core by $\Re^t_{peers}$. Since, our WCET analysis assumes that the tasks are aligned to the beginning of a bus slot, during *LatestFinish* time computation, this alignment cost needs to be considered. In the worst case, all of the peers of a task and the task itself may encounter maximum alignment cost (equals B). Thus the *LatestFinish* time is defined as

follows:

$$\begin{aligned} LatestFinish(t) \quad &= LatestReady(t) + WCET(t) \\ &+ \sum_{t^c \in \Re^t_{peers}} WCET(t^c) \\ &+ (|\Re^t_{peers}| + 1) \times B \end{aligned} \tag{6.4}$$

Here $|\Re^t_{peers}|$ represents the number of peers of task t. This approach keeps our framework highly modular since the WCRT computation can be carried out given the WCET and BCET values of each task t, and without knowing their worst/best case starting time. Finally WCRT of an application is defined as follows:

$$WCRT = \max_t(LatestFinish(t)) - \min_t(EarliestReady(t)) \tag{6.5}$$

that is, the duration from the earliest start time of any task to the latest completion time of any task.

Our WCRT analysis framework is shown in Figure 6.3. Initially a task t' cannot overlap (that is, *interfere*) with a task t if and only if 1) task t' depends on t and vice versa by the partial order imposed from the task graph or 2) t and t' execute on the same core (by virtue of non-preemptive execution). After the WCRT analysis, new interference information is generated if two independent tasks which accounted for shared cache conflicts in the cache analysis are found to have non-overlapping lifetimes, that is, their *[EarliestReady(t),LatestFinish(t)]* intervals do not overlap. This new interference information is again fed to the shared cache conflict analysis module which may further tighten several tasks' WCET in presence of shared bus. This process continues until the interference among all the tasks stabilizes. In the following, we shall prove the termination of our WCRT analysis technique.

Observation 6.3.1. *T_1 and T_2 are beginning time of any two bus slots in the round robin schedule belonging to some core n. Say the worst case cost of a program running on core n, if started after δ time from T_1 is W_1 and if started at same δ time after T_2 is W_2. Then, always W_1=W_2.*

Theorem 6.3.1. *For any task t, its BCET and EarliestReady(t) do not change across different iterations of L2 cache conflict and WCRT analysis.*

Proof. First we shall prove that the best-case *hit-miss* classification of an instruction in L1 and L2 cache does not change across different iterations of L2 cache conflict analysis. Level 2

cache conflict analysis from [15] only changes the memory blocks classified as "Always Hit" in L2 cache to "Non-Classified" due to interference from conflicting tasks. An "Always Hit" memory block in L2 cache should have "Always Miss" or "Non-Classified" status in L1 cache. A memory block classified as L1 "Always Miss" is considered as L2 cache hit in the best case irrespective of whether it is AH or NC in L2 cache. Similarly, a "Non-Classified" memory block in L1 is considered as L1 cache hit in the best case irrespective of its classification in the L2 cache. Hence, L2 cache conflict analysis cannot change the best-case fixed latency of a memory reference.

We prove that for same $EarliestReady(t)$ of a task t, its BCET does not change across different iterations of bus analysis. We have already proved that best-case fixed latency of a memory reference cannot change in different iterations of WCRT analysis. Hence a memory reference will encounter different latency if and only if its corresponding waiting time for bus access is different. Waiting time for the bus, in turn will be different if and only if the starting time of the corresponding memory reference is different. In bus analysis, we always assume all loops inside a task start at time 0 irrespective of the task's original starting time. Essentially, we compute the best case cost of a loop independent of the starting time of the task. Hence the best case cost of loops do not change across different iterations of the analysis. Only the set of L1 cache misses outside outer-most loops may encounter different bus delay for different starting time and thus leading to a different BCET value. But starting time of all those memory references may change if and only if the $EarliestReady(t)$ of the task t changes. Hence we derive, for the same $EarliestReady(t)$ of a task t, its BCET does not change across different iterations of L2 cache conflict and WCRT analysis.

We prove that $EarliestReady(t)$ does not change through contradiction. Let us assume that for a task t, its $EarliestReady(t)$ changes. This must be due to a change in its predecessor's $EarliestReady$ time because a task's BCET remains unchanged for the same $EarliestReady$ time. Proceeding backwards, $EarliestReady(src)$ must have changed where src is a task without any predecessor, contradicting the fact that $EarliestReady(src) = 0$. Hence, for a task t its $EarliestReady(t)$ does not change. □

Theorem 6.3.2. *Task interferences monotonically decrease (strictly decrease or remain the same) across different iterations of our analysis framework (Figure 6.3).*

Proof. We prove by induction on number of iterations.

Base Case: In the first iteration, tasks are assumed to conflict with all the tasks on other cores (except those excluded by partial order). This is the worst case task interference scenario. Thus, the task interferences of the second iteration definitely monotonically decrease compared to the first iteration.

Induction Step: We need to show that the task interferences monotonically decrease from iteration n to iteration $n+1$ assuming that the task interferences monotonically decrease from iteration $n-1$ to n. We prove by contradiction. Assume two tasks i and j do not interfere at iteration n, but interfere at iteration $n+1$. There are two cases.

- $EarliestReady(j) \geq LatestFinish(i)$ at iteration n, but at iteration $n+1$, we obtain $EarliestReady(j) < LatestFinish(i)$. This implies that $LatestFinish(i)$ at iteration $n+1$ increases because $EarliestReady(j)$ remains unchanged across iterations according to Theorem 6.3.1. $LatestFinish(i)$ at iteration $n+1$ can change due to three reasons: (1) at iteration $n+1$, the WCET of the task i itself increases; (2) the WCET of some tasks which task i depends on directly on indirectly increases; and (3) the WCET of some tasks increases as a result of which either the number of peers of task i ($|\Re^t_{peers}|$) increases or the WCET of a peer of task i increases. In summary, at least one task's WCET is increased. The WCET increase of some task at iteration $n+1$ can be for two reasons: (1) More memory blocks are changed from Always Hit to Non-Classified due to the task interference increase at iteration n; (2) the task is started at some different offset from the beginning of a bus slot — as a result bus analysis may increase the WCET. Since we always align the task to the beginning of a bus slot belonging to the core, Observation 6.3.1 rules out the second option. First option contradicts with the assumption that task interferences monotonically decrease at iteration n.

- $EarliestReady(i) \geq LatestFinish(j)$ at iteration n, but at iteration $n+1$, we obtain $EarliestReady(i) < LatestFinish(j)$. The proof is symmetric to the first case.

□

6.4 Experimental evaluation

Experimental setup We compile our benchmarks for SimpleScalar Portable instruction set (PISA) [81] – a MIPS like instruction set architecture. The individual tasks are compiled into

Table 6.1: Description of Benchmarks used

Benchmark	*Description*	*Bytes*	*Lines of Code*
matmult	Matrix multiplication	3737	163
jfdcint	Discrete-cosine transformation	16028	375
adpcm	Adaptive pulse code modulation algorithm	26852	879
edn	Implements the jpegdct algorithm together with other signal processing algorithms	10563	285
fft	Fast Fourier Transform	6244	219
fir	Finite impulse response filter (signal processing algorithms)	11965	276
compress	Data compression program	13411	508
statemate	Automatically generated code by the STAtechart Real-time-Code generator STARC	52618	1276

SimpleScalar PISA compliant binaries, and their control flow graphs (CFGs) are extracted as input to the analysis framework. Each processor core has an in-order pipeline along with an instruction cache.

In order to experimentally evaluate the accuracy of our analysis results, we have built a cycle-accurate simulation infrastructure on top of the CMP-SIM simulator — a multi-core extension of simplescalar toolset. We have extended CMP-SIM in three ways. First, we extend CMP-SIM to handle PISA binaries. Secondly, CMP-SIM models shared caches, but not shared bus across cores. We have extended CMP-SIM with shared bus modeling to provide a cycle-accurate simulator for present day multi-core architectures; we instrumented each L1 cache miss in the simulation to go through a round-robin TDMA based shared bus. Finally, the existing CMP-SIM simulator only allows simulation of independent programs, each running on a different core. We have extended CMP-SIM to simulate concurrent applications represented as task graphs.

All experiments are performed on a 3 GHz Pentium 4 machine having 1 GB of RAM and running Ubuntu Linux 8.10 as the operating system.

Analysis of independent programs First we analyze independent programs running on multiple cores. Later we present results from analyzing the task graph from a real-life space debris monitoring program.

We have performed experiments both for two-core and four-core platforms with the following cache configuration: L1 cache hit latency = 1 cycle, L2 cache hit latency = 6 cycles and memory latency = 30 cycles. Each private L1 cache is direct-mapped and has a size of 1 KB with block size of 32 bytes. The shared L2 cache is 4-way associative and has a size of 2 KB with block size of 64 bytes. The shared bus connecting the different cores is TDMA based and is

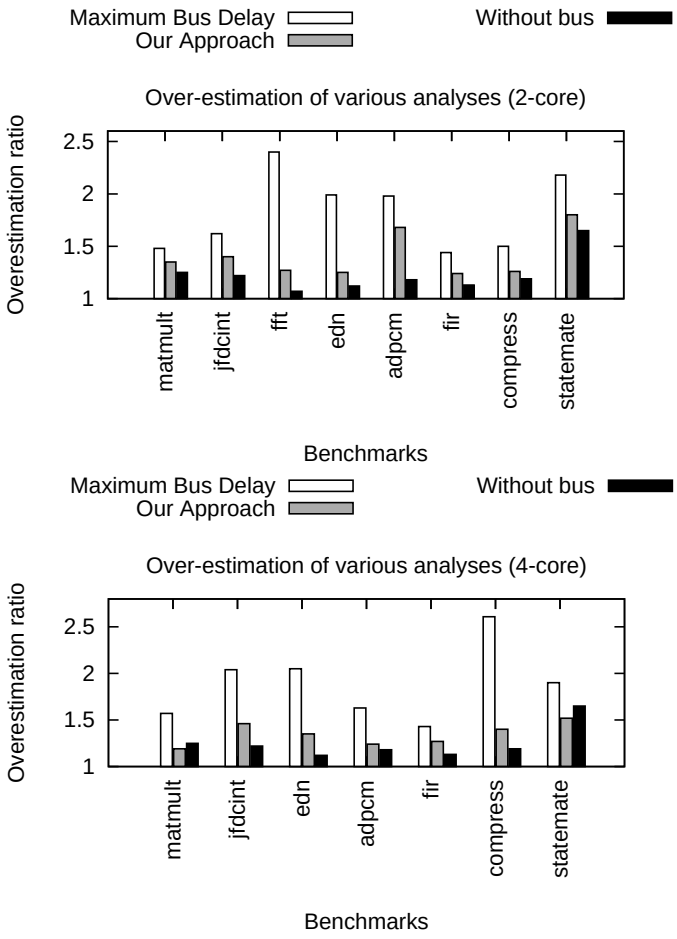


Figure 6.5: Overestimation in WCET analysis

accessed by all the cores in a round-robin fashion. Each core has a bus slot length of 80 cycles.

For analyzing independent programs running on different cores, we have chosen benchmarks from [2]. The set of benchmarks is described in Table 6.1. We have benchmarks with small/medium code size (e.g., matmult, fft) as well as large code size (e.g., statemate, adpcm). Also, our chosen set of benchmarks contain both single-path programs (e.g., matmult, jfdcint) and multiple-path programs (e.g., compress, statemate). We have chosen a long running program statemate as a representative to run on a single core (to increase the probability of interference in shared L2 instruction cache) and different combinations of other programs are run on other cores. For all experiments on 2-cores, the estimation results for program X (other than statemate) correspond to running X in one core and statemate in the other core. The reported results for statemate are the average of all the runs. For 4-core experiments we either run (edn, adpcm, compress, statemate) on the 4 cores, or we run (matmult, fir, jfdcint, statemate) on the 4 cores. The reported results for statemate are the average of what we get from running (edn,

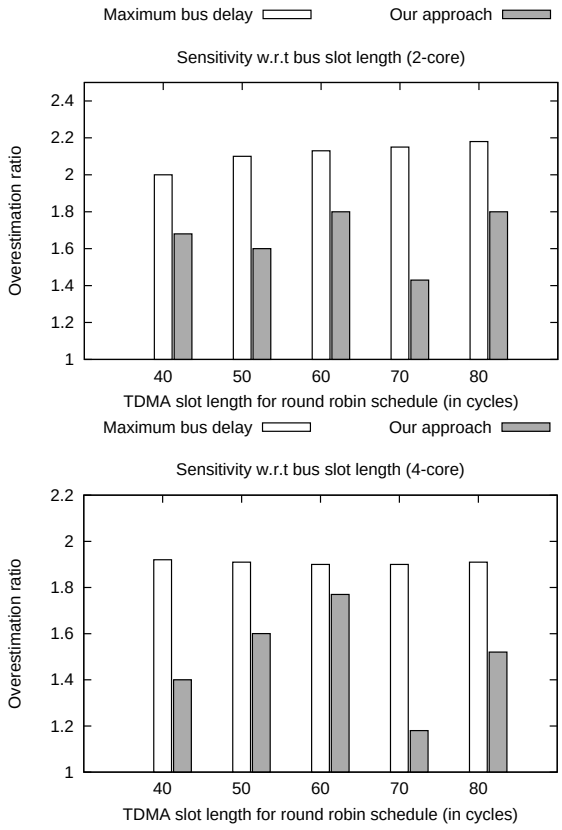


Figure 6.6: Sensitivity of WCET analysis with bus slot length

adpcm, compress, statemate) and (matmult, fir, jfdcint, statemate).

Figure 6.5 demonstrates the precision of our analysis. The overestimation ratio in Figure 6.5 is computed by dividing the estimated WCET with the observed WCET. Observed WCET of the program is computed through simulation by running it on a few sample inputs and taking the maximum of these running times. Our analysis results are marked by "Our Approach" in Figure 6.5.

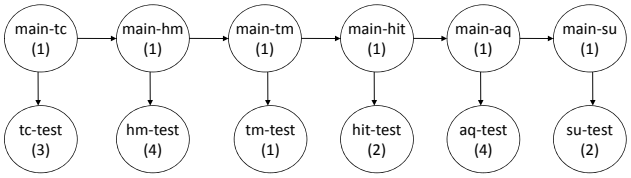
To compare the effect of shared bus analysis in WCET estimation, we have also shown the overestimation ratio for an architecture without shared bus (shown by the bar "Without Bus" in Figure 6.5). Some of the cases show a decreased overestimation ratio (e.g., statemate in 4-core) in presence of shared bus. Note that observed WCET also increases in presence of shared bus. Therefore, a decreased overestimation ratio signifies that the overestimation is more due to the presence of other micro-architectural entities (i.e., pipeline, cache) than due to the shared bus.

Due to the difficulty of analyzing the shared bus, an obvious solution is to always assume the worst case waiting time for bus access for every memory reference that may potentially access the shared bus. Overestimation introduced from this approach is also depicted in Figure

6.5 as “Maximum Bus Delay”. As shown, for reasonably large programs (e.g., edn, fft, adpcm, statemate) the overestimation is excessive, making this approach not useful in practice.

As shown in Figure 6.5, our analysis can produce tight WCET estimates. The average overestimation from our approach is around 40%. For single path programs (e.g., matmult, edn), the overestimation is within 35%.

Figure 6.6 presents the over-estimation ratio of benchmark statemate in various analysis results (“Our approach”, “Maximum bus delay” as mentioned above) for different bus slot lengths. Figure 6.6 shows that the precision of our analysis depends on the bus slot length. However, the over-estimation is much tighter than the analysis which uses maximum bus delay for each bus transaction (“Maximum bus delay” approach). Figure 6.6 also shows clearly that our analysis is not tied with a particular bus slot length.



Core	Task name	Code size (bytes)
1	main-tc	240
1	main-hm	240
1	main-tm	240
1	main-hit	240
1	main-aq	240
1	main-su	240

Core	Task name	Code size (bytes)
1	tm-test	56,960
2	hit-test	10,776
2	su-test	50,176
3	tc-test	45,368
4	hm-test	44,176
4	aq-test	44,128

Figure 6.7: DEBIE task graph and task sizes

Analysis of task graphs in DEBIE To evaluate our WCRT framework we have analyzed a large fragment of a real life DEBIE program [82], an in-situ space debris monitoring program. The task graph for the fragment of DEBIE program is given in Figure 6.7. We consider a system with four cores. The number inside each task of the task graph shows the mapping of the tasks to the processor cores. Codesize of each task is given in Figure 6.7.

For the experiments with DEBIE, private L1 caches are changed to 2-way associative, 2 KB caches and the shared L2 cache is changed to a 4-way associative, 8 KB cache. The reason for changing the above-mentioned parameters is the relatively large code-sizes of the tasks in the DEBIE benchmark. Without a larger instruction cache, both simulation and estimation encounter cache thrashing making it difficult to evaluate the accuracy of our analysis. For this

reason the L2 cache size is increased to 8KB in these experiments. All other parameters (bus slot, cache line size, cache hit latency, cache miss latency) remain unchanged from the settings used previously.

The analysis results are shown in Table 6.2. The value sim_{bus} denotes the observed WCRT (maximum execution time obtained from cycle-accurate simulation on a few inputs in presence of shared cache and bus). The values $wcrt_{max\_bus\_delay}$ and $wcrt_{ours}$ denote the WCRT estimates taking the maximum bus delay for every bus access and WCRT estimate from our analysis respectively — all in presence of shared cache and bus.

sim_{bus}	$wcrt_{max\_bus\_delay}$	$wcrt_{ours}$
50432	192567	61997

Table 6.2: Results from DEBIE ($\times 10^4$ cycles)

We observe that the overestimation coming through our analysis is only 22.9% when compared to the simulation results (compare $wcrt_{ours}$ with sim_{bus}). Our analysis is time-efficient. The time to produce $wcrt_{ours}$ is less than 1.5 minutes. This time includes the full analysis time – starting from intra-core analysis to the end of our iterative and combined shared cache and bus analysis.

6.5 Extensions

Other multi-processor architectures: Our analysis can easily be adopted with minimal changes for other kind of architectures featuring shared cache and shared bus. For example, consider the multi-processor architecture shown in Figure 6.8. In this type of architectures, a processor chip has multiple cores (as shown by Core 0, . . ., Core N in Figure 6.8). Each core in the processor has an on-chip L1 cache and all cores share an on-chip L2 cache through a crossbar switch. There might be multiple processor chips in the architecture (as shown by Processor 0 and Processor 1) and they access the off-chip system memory through an off-chip shared bus. Intel's dual-processor and dual core architecture [83] are similar to the one shown in Figure 6.8 where each processor has 2 cores.

Our analysis framework can easily be tuned to work with the above mentioned architectures. Each core can still be analyzed separately to produce per-core analysis result. Later, the interference between all the tasks running on the *same* chip can be used in on-chip shared cache conflict analysis. However, we observe that only shared L2 cache misses appear in the off-chip bus. Therefore, *only* shared L2 cache misses encounter variable amount of latency. On the other

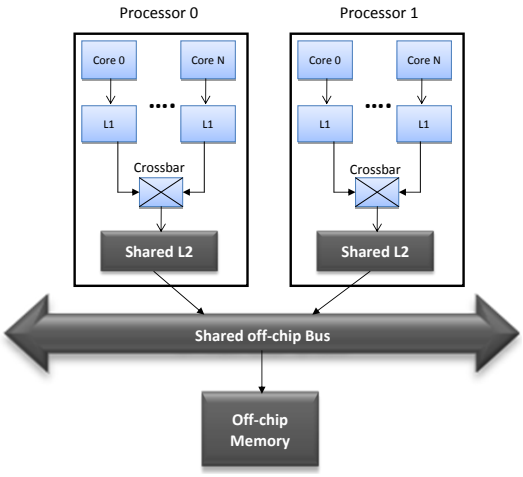


Figure 6.8: A multi-processor architecture featuring on-chip shared L2 cache

way, our bus analysis has to be employed only for shared L2 cache misses (which is a subset of L1 cache misses) instead of all L1 cache misses.

Applications using shared library: In our current implementation, all shared libraries are copied inside each task. However, our analysis can easily be extended with shared libraries as follows: first, a single shared memory block in L2 cache could be accessed by two independent tasks. In this case, these two accesses are not conflicting and both accesses are *cache hits*. However, our shared L2 cache conflict analysis will interpret these two accesses as conflicting and in the worst-case both accesses to this shared memory block (in two different tasks) will be estimated as *cache misses*. Our shared L2 cache conflict analysis can easily be changed to incorporate this information. Second, in presence of shared libraries, initial cache state of a task may contain some shared memory blocks left by its predecessor tasks. This initial cache state can be estimated by taking into account all possible dependencies imposed by the application task graph.

Other application model: In this work, we use a non-preemptive execution model. Using a preemptive execution model requires us to analyze the delay due to the preemptions. The most important problem is to analyze the amount of delay introduced for reloading the cache when preempted task restarts execution. This reloading cost is known as *cache related preemption delay* (CRPD). Existing works produce an upper bound on CRPD so that it could be added to the actual WCET of the task. In the presence of shared bus, the waiting time to access the shared bus depends on the exact timestamp of each memory reference; this in turn is affected by the possible points where preemption can take place and the associated CRPD at that points. We plan to model the effect of CRPD on our analysis in the future.

6.6 Chapter summary

This chapter presents an integrated analysis framework that considers the timing effects generated by both the shared cache and the shared bus. Existing works had concentrated on the modeling of either shared cache or shared bus, but not both. This chapter described how the timing interactions of shared cache and shared bus can affect the overall response time of an application. Moreover, we have presented a new and efficient TDMA shared bus analysis technique that avoids virtual loop unrolling. Our experimental results are compared with the cycle-accurate simulation results to evaluate the precision of our analysis. We have also discussed that our analysis can be applied to similar multi-processor architectures.

Chapter 7

A Unified WCET Analysis Framework for Multi-core Platforms

So far in this dissertation, we have discussed the modeling of shared resources in multi-core (shared cache and shared bus) and the timing interactions created among these shared resources. In this Chapter, we shall show the major challenges in WCET analysis while the shared resources in multi-core interact with other basic micro-architectural components (such as pipelines and branch predictors). Our work in this chapter is dedicated to build a *sound* WCET analysis framework for multi-core, which is capable to provide safe WCET estimates in the presence of advanced micro-architectural features (*e.g.* out-of-order and superscalar pipelines, speculative execution and so on).

7.1 Introduction

We have seen that the WCET analysis in multi-core becomes challenging due to the presence of shared caches and shared buses. The presence of a shared cache requires the modeling of inter-core cache conflicts. On the other hand, the presence of a shared bus introduces variable bus access latency to accesses to shared cache and shared main memory. The delay introduced by shared cache conflict misses and shared bus accesses is propagated by different pipeline stages of the processor and affects the overall execution time of a program. WCET analysis is further complicated by a commonly known phenomenon called *timing anomalies* [16]. In the presence of timing anomalies, a local worst case scenario may not lead to the WCET of the overall program. As an example, a *cache hit* rather than a *cache miss* may lead to the WCET of

the entire program. Therefore, we cannot always assume a *cache miss* or *maximum bus delay* as the worst case scenario, as the assumptions are not just *imprecise*, but they may also lead to an *unsound* WCET estimation. Existing works have proposed to model the shared cache and/or the shared bus ([35; 15; 40; 41; 42]) in isolation, but all of these previous solutions ignore the interactions of shared resources with important micro-architectural features such as pipelines and branch predictors.

In this Chapter, we propose a WCET analysis framework for multi-core platforms featuring both a shared cache and a shared bus. In contrast to existing work, our analysis can efficiently model the interaction of the shared cache and bus with different other micro-architectural features (*e.g.* pipeline, branch prediction). A few such meaningful interactions include the effect of shared cache conflict misses and shared bus delays on the pipeline, the effect of speculative execution on the shared cache etc. Moreover, our analysis framework does not rely on a *timing-anomaly free* architecture and gives a *sound* WCET estimate even in the presence of timing anomalies. In summary, the central contribution of this paper is to propose a unified analysis framework that features most of the basic micro-architectural components (pipeline, (shared) cache, branch prediction and shared bus) in a multi-core processor.

Our analysis framework deals with timing anomalies by representing the timing of each pipeline stage as an *interval*. The interval covers all possible latencies of the corresponding pipeline stage. The latency of a pipeline stage may depend on cache miss penalties and shared bus delays. On the other hand, cache and shared bus analysis interact with the pipeline stages to compute the possible latencies of a pipeline stage. Our analysis is context sensitive — it takes care of different procedure call contexts and different micro-architectural contexts (*i.e.* cache and bus) when computing the WCET of a single basic block. Finally, WCET of the entire program is formulated as an integer linear program (ILP). The formulated ILP can be solved by any commercial solver (*e.g.* CPLEX) to get the whole program's WCET.

We have implemented our framework in an extended version of Chronos [23], a freely available, open-source, single-core WCET analysis tool. To evaluate our approach, we have also extended a *cycle-accurate simulator* [81] with both shared cache and shared bus support. Our experiments with moderate to large size benchmarks from [2] show that we can obtain tight WCET estimates for most of the benchmarks in a wide range of micro-architectural configurations.

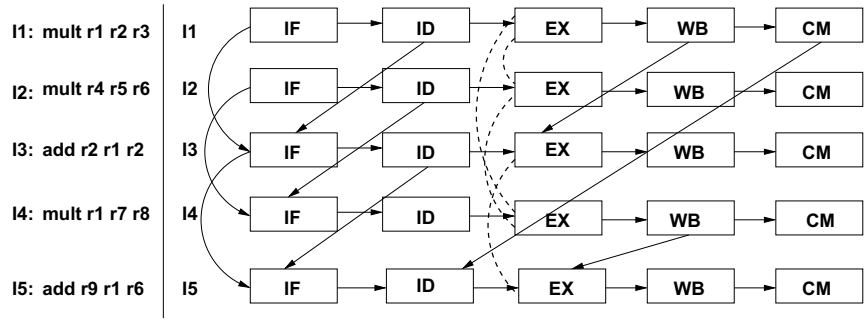


Figure 7.1: Execution graph for the example program in a 2-way superscalar processor with 2-entry instruction fetch queue and 4-entry reorder buffer. Solid edges show the dependency between pipeline stages, whereas the dotted edges show the contention relation

7.2 Background

In this section, we introduce the basic background behind our WCET analysis framework. Our WCET analysis framework for multi-core is based on the pipeline modeling of [17].

Pipeline modeling through execution graphs The central idea of pipeline modeling revolves around the concept of the *execution graph* [17]. The execution graph is constructed for each basic block in the program control flow graph (CFG). For each instruction in the basic block, the corresponding execution graph contains a node for each of the pipeline stages. We assume a five stage pipeline — *instruction fetch* (IF), *decode* (ID), *execution* (EX), *write back* (WB) and *commit* (CM). Edges in the execution graph capture the dependencies among pipeline stages; either due to resource constraints (instruction fetch queue size, reorder buffer size etc.) or due to data dependency (*read after write hazard*). The timing of each node in the execution graph is represented by an interval, which covers all possible latencies suffered by the corresponding pipeline stage.

Figure 7.1 shows a snippet of assembly code and the corresponding execution graph. The example assumes a 2-way superscalar processor with 2-entry instruction fetch queue (IFQ) and 4-entry reorder buffer (ROB). Since the processor is a 2-way superscalar, instruction I3 cannot be fetched before the fetch of I1 finishes. This explains the edge between IF nodes of I1 and I3. On the other hand, since IFQ size is 2, IF stage of I3 cannot start before ID stage of I1 finishes (edge between ID stage of I1 and IF stage of I3). Note that I3 is data dependent on I1 and similarly, I5 is data dependent on I4. Therefore, we have edges from WB stage of I1 to EX stage of I3 and also from WB stage of I4 to EX stage of I5. Finally, as ROB size is 4, I1 must

be removed from ROB (*i.e.* committed) before I5 can be decoded. This explains the edge from CM stage of I1 to ID stage of I5.

A dotted edge in the execution graph (*e.g.* the edge between EX stage of I2 and I4) represents contention relation (*i.e.* a pair of instructions which may contend for the same functional unit). Since I2 and I4 may contend for the same functional unit (multiplier), they might delay each other due to contention. The pipeline analysis is iterative. Analysis starts without any timing information and assumes that all pairs of instructions which use same functional units and can coexist in the pipeline, may contend with each other. In the example, therefore, the analysis starts with {(I1,I2), (I2,I4), (I1,I4), (I3,I5)} in the contention relation. After one iteration, the timing information of each pipeline stage is obtained and the analysis may rule out some pairs from the contention relation if their timing intervals do not overlap. With this updated contention relation, the analysis is repeated and subsequently, a refined timing information is obtained for each pipeline stage. Analysis is terminated when no further elements can be removed from the contention relation. WCET of the code snippet is then given by the worst case completion time of the CM node for I5.

7.3 Overview of our analysis

Figure 10.2 gives an overview of our analysis framework. Each processor core is analyzed at a time by taking care of the *inter-core conflicts* generated by all other cores. Figure 10.2 shows the analysis flow for some program A running on a dedicated processor core. The overall analysis can broadly be classified into two separate phases: 1) micro-architectural modeling and 2) path analysis. In micro-architectural modeling, the timing behavior of different hardware components is analyzed (as shown by the big dotted box in Figure 10.2). We use abstract interpretation (AI) based cache analysis [9] to categorize memory references as all-hit (AH) or all-miss (AM) in L1 and L2 cache. A memory reference is categorized AH (AM) if the resulting access is always a *cache hit* (*miss*). If a memory reference cannot be categorized as AH or AM, it is categorized as *unclassified* (NC). In the presence of a shared L2 cache, categorization of a memory reference may change from AH to NC due to the *inter-core conflicts* [15]. Moreover, as shown in Figure 10.2, L1 and L2 cache analysis has to consider the effect of *speculative execution* when a branch instruction is mispredicted (refer to Section 7.6 for details). Similarly, the timing effects generated by the mispredicted instructions are also taken into account during

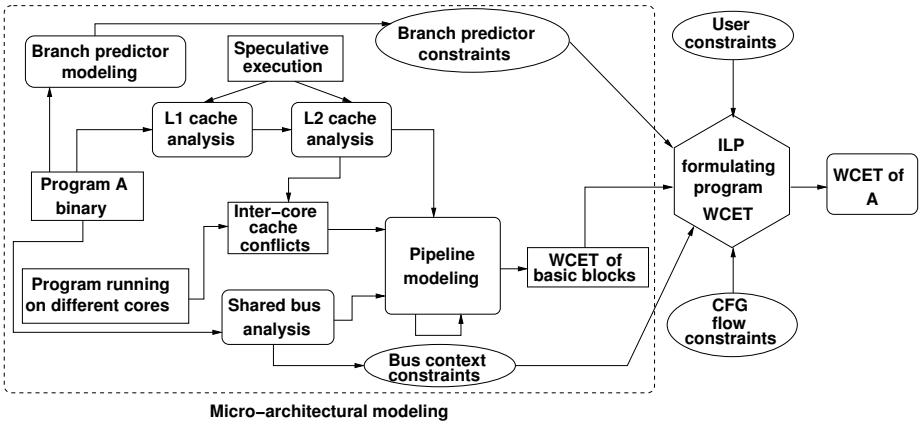


Figure 7.2: Overview of our analysis framework

the iterative pipeline modeling (refer to [17] for details). The *shared bus analysis* computes the bus context under which an instruction can execute. The outcome of cache analysis and shared bus analysis is used to compute the latency of different pipeline stages during the analysis of the pipeline (refer to Section 7.4 for details). Pipeline modeling is iterative and it finally computes the WCET of each basic block. WCET of the entire program is formulated as *maximizing* the objective function of a *single integer linear program* (ILP). WCETs of individual basic blocks are used to construct the objective function of the formulated ILP. The constraints of the ILP are generated from the structure of the program's control flow graph (CFG), micro-architectural modeling (branch predictor and shared bus) and additional user-given constraints (*e.g.* loop bounds). The modeling of the branch predictor generates constraints to bound the execution count of mispredicted branches (for details refer to [18]). On the other hand, constraints generated for bus contexts bound the execution count of a basic block under different bus contexts (for details, refer to Section 7.5). *Path analysis* finds the longest feasible program path from the formulated ILP through *implicit path enumeration* (IPET). Any ILP solver (*e.g.* CPLEX) can be used for IPET and for deriving the whole program's WCET.

System and application model We assume a multi-core processor with each core having a private L1 cache. Additionally, multiple cores share a L2 cache. The extension of our framework for more than two levels of caches is straightforward. If a memory block is not found in L1 or L2 cache, it has to be fetched from the main memory. Any memory transaction to L2 cache or main memory has to go through a shared bus. For shared bus, we assume a *TDMA-based round robin arbitration policy*, where a fixed length bus slot is assigned to each core. We also assume fully

separated caches and buses for instruction and data memory. Therefore, the data references do not interfere with the instruction references. In this work, we only model the effect of instruction caches. However, the data cache effects can be considered in a similar fashion. Since we consider only instruction caches, the cache miss penalty (computed from cache analysis) directly affects the *instruction fetch* (IF) stage of the pipeline. We do not consider *self modifying* code and therefore, we do not need to model the coherence traffic. Finally, we consider the *LRU cache replacement policy* and *non-inclusive* caches only. Later in Section 7.10, we shall extend our framework for FIFO cache replacement policy and we shall also discuss the extension of our framework for other cache replacement policies (*e.g.* PLRU) and other cache hierarchies (*e.g.* inclusive).

7.4 Interaction of shared resources with pipeline

Let us assume each node i in the execution graph is annotated with the following timing parameters, which are computed iteratively:

- $earliest[t_i^{ready}], earliest[t_i^{start}], earliest[t_i^{finish}]$: Earliest ready, earliest start and earliest finish time of node i, respectively.
- $latest[t_i^{ready}], latest[t_i^{start}], latest[t_i^{finish}]$: Latest ready, latest start and latest finish time of node i, respectively.

For each pipeline stage i, $earliest[t_i^{ready}]$ and $earliest[t_i^{start}]$ are initialized to *zero*, whereas, $earliest[t_i^{finish}]$ is initialized to the *minimum latency* suffered by the pipeline stage i. On the other hand, $latest[t_i^{ready}]$, $latest[t_i^{start}]$ and $latest[t_i^{finish}]$ are all initialized to ∞ for each pipeline stage i. The active time span of node i can be captured by the following timing interval: $[earliest[t_i^{ready}], latest[t_i^{finish}]]$. Therefore, each node of the execution graph is initialized with a timing interval $[0, \infty]$.

Pipeline modeling is iterative. The iterative analysis starts with the coarse interval $[0, \infty]$ for each node and subsequently, the interval is tightened in each iteration. The computation of a precise interval takes into account the analysis result of caches and shared bus. The iterative analysis eliminates certain infeasible contention among the pipeline stages in each iteration, thereby leading to a tighter timing interval after each iteration. The iterative analysis starts with a contention relation. Such a contention relation contains pairs of instructions which may

potentially delay each other due to contention. Initially, all possible pairs of instructions are included in the contention relation and after each iteration, pairs of instructions whose timing intervals do not overlap, are removed from this relation. If the contention relation does not change in some iteration, the iterative analysis terminates. Since the number of instructions in a basic block is finite, the contention relation contains a finite number of elements and in each iteration, at least one element is removed from the relation. Therefore, this analysis is guaranteed to terminate. Moreover, if the contention relation does not change, the timing interval of each node reaches a fixed-point after the analysis terminates. In the following, we shall discuss how the presence of a shared cache and a shared bus affects the timing information of different pipeline stages.

7.4.1 Interaction of shared cache with pipeline

Let us assume $CHMC_i^{L1}$ ($CHMC_i^{L2}$) denotes the AH/AM/NC cache *hit-miss* classification of an IF node i in L1 (shared L2) cache. Further assume that E_i denotes the possible latencies of an *IF* node i without considering any shared bus delay. E_i can be defined as follows:

$$E_i = \begin{cases} 1, \text{ if } CHMC_i^{L1} = AH; \\ LAT^{L1} + 1, \quad \text{if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AH; \\ LAT^{L1} + LAT^{L2} + 1, \quad \text{if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AM; \\ [LAT^{L1} + 1, LAT^{L1} + LAT^{L2} + 1], \quad \text{if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = NC; \\ [1, LAT^{L1} + 1], \quad \text{if } CHMC_i^{L1} = NC \wedge CHMC_i^{L2} = AH; \\ [1, LAT^{L1} + LAT^{L2} + 1], \text{ otherwise.} \end{cases} \tag{7.1}$$

where LAT^{L1} and LAT^{L2} represent the fixed L1 and L2 cache miss latencies respectively. Note that the interval-based representation captures the possibilities of both a *cache hit* and a *cache miss* in case of an NC categorized cache access. Therefore, the computation of E_i can also deal with the architectures that exhibit timing anomalies.

7.4.2 Interaction of shared bus with pipeline

Let us assume that we have a total of $\mathcal{C}$ cores and the TDMA-based round robin scheme assigns a slot length S_l to each core. Therefore, the length of one complete *round* is $S_l\mathcal{C}$. We begin with the following definitions which are used throughout the paper:

Definition 7.4.1. *(*TDMA offset*) A* TDMA offset *at a particular time T is defined as the relative distance of T from the beginning of the last scheduled* round*. Therefore, at time T, the TDMA offset can be precisely defined as $T \ mod \ S_l\mathcal{C}$.*

Definition 7.4.2. *(*Bus context*) A* Bus context *for a particular execution graph node i is defined as the set of TDMA offsets reaching/leaving the corresponding node. For each execution graph node i, we track the incoming bus context (denoted O_i^{in}) and the outgoing bus context (denoted O_i^{out}).*

For a task executing in core p (where $0 \leq p < \mathcal{C}$), $latest[t_i^{finish}]$ and $earliest[t_i^{finish}]$ are computed for an IF execution graph node i as follows:

$$latest[t_i^{finish}] = latest[t_i^{start}] + max\_lat_p(O_i^{in}, E_i) \tag{7.2}$$

$$earliest[t_i^{finish}] = earliest[t_i^{start}] + min\_lat_p(O_i^{in}, E_i) \tag{7.3}$$

Note that max_lat_p, min_lat_p are not constants and depend on the incoming bus context (O_i^{in}) and the set of possible latencies of IF node i (E_i) in the absence of a shared bus. max_lat_p and min_lat_p are defined as follows:

$$max\_lat_p(O_i^{in}, E_i) = \begin{cases} 1, \ \text{ if } CHMC_i^{L1} = AH; \\ \\ \max\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), \ \text{ otherwise.} \end{cases} \tag{7.4}$$

$$min\_lat_p(O_i^{in}, E_i) = \begin{cases} 1, \ \text{ if } CHMC_i^{L1} \neq AM; \\ \\ \min\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), \ \text{ otherwise.} \end{cases} \tag{7.5}$$

In the above, E_i represents the set of possible latencies of an *IF* node i in the absence of shared bus delay (refer to Equation 7.1). Given a TDMA offset o and latency t in the absence of shared bus delay, $\Delta_p(o,t)$ computes the total delay (including shared bus delay) faced by the IF stage

of the pipeline. $\Delta_p(o,t)$ can be defined as follows (similar to [40] or [41]):

$$\Delta_p(o,t) = \begin{cases} t, & \text{if } pS_l \leq o+t \leq (p+1)S_l; \\ t + pS_l - o, & \text{if } o < pS_l; \\ t + (\mathcal{C}+p)S_l - o, & \text{otherwise.} \end{cases} \tag{7.6}$$

In the following, we shall now show the computation of incoming and outgoing bus contexts (*i.e.* O_i^{in} and O_i^{out} respectively) for an execution graph node i.

Computation of O_i^{out} from O_i^{in} The computation of O_i^{out} depends on O_i^{in}, on the possible latencies of execution graph node i (including shared bus delay) and on the contention suffered by the corresponding pipeline stage. In the modeled pipeline, inorder stages (*i.e.* IF, ID, WB and CM) do not suffer from contention. But the out-of-order stage (*i.e.* EX stage) may experience contention when it is *ready* to execute (*i.e.* operands are available) but cannot *start* execution due to the unavailability of a functional unit. Worst case contention period of an execution graph node i can be denoted by the term $latest[t_i^{start}] - latest[t_i^{ready}]$. For *best case* computation, we conservatively assume the absence of contention. Therefore, for a particular core p ($0 \leq p < \mathcal{C}$), we compute O_i^{out} from the value of O_i^{in} as follows:

$$O_i^{out} = \begin{cases} u(O_i^{in}, E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]), & \text{if } i = EX; \\ u(O_i^{in}, \bigcup_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t)), & \text{if } i = IF; \\ u(O_i^{in}, E_i), & \text{otherwise.} \end{cases} \tag{7.7}$$

Here, u denotes the update function on TDMA offset set with a set of possible latencies of node i and is defined as follows:

$$u(O,X) = \bigcup_{o \in O, t \in X} \{(o+t)\ mod\ S_l\mathcal{C}\} \tag{7.8}$$

Note that $E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]$ captures all possible latencies suffered by the execution graph node i, taking care of contentions as well. Therefore, O_i^{out} captures all possible TDMA offsets exiting node i, when the same node is entered with bus context O_i^{in}. More precisely, assuming that O_i^{in} represents an over-approximation of the incoming bus context at node i, the computation by Equation 7.7 ensures that O_i^{out} represents an over-approximation of

the outgoing bus context from node i.

Computation of O_i^{in} The value of O_i^{in} depends on the value of O_j^{out}, where j is a predecessor of node i in the execution graph. If $pred(i)$ denotes all the predecessors of node i, clearly, $\cup_{j \in pred(i)} O_j^{out}$ gives a *sound* approximation of O_i^{in}. However, it is important to observe that not all predecessors in the execution graph can propagate TDMA offsets to node i. Recall that the edges in the execution graph represent dependency (either due to resource constraints or due to true data dependences). Therefore, node i in the execution graph can only *start* when all the nodes in $pred(i)$ have *finished*. Consequently, the TDMA offsets are propagated to node i only from the predecessor j, which finishes *immediately* before i is ready. Nevertheless, our static analyzer may not be able to compute a single predecessor that propagates TDMA offsets to node i. However, for two arbitrary execution graph nodes $j1$ and $j2$, if we can guarantee that $earliest[t_{j2}^{finish}] > latest[t_{j1}^{finish}]$, we can also guarantee that $j2$ finishes *later* than $j1$. The computation of O_i^{in} captures this property:

$$O_i^{in} = \bigcup \{O_j^{out} \mid j \in pred(i) \wedge\ earliest[t_{pmax}^{finish}] \leq latest[t_j^{finish}]\} \tag{7.9}$$

where $pmax$ is a predecessor of i such that $latest[t_{pmax}^{finish}] = \max_{j \in pred(i)} latest[t_j^{finish}]$. Therefore, O_i^{in} captures all possible outgoing TDMA offsets from the predecessor nodes that are *possibly* finished *latest*. Given that the value of O_j^{out} is an over-approximation of the outgoing bus context for each predecessor j of i, Equation 7.9 gives an over-approximation of the incoming bus context at node i. Finally, Equation 7.7 and Equation 7.9 together ensure a sound computation of the bus contexts at the entry and exit of each execution graph node.

7.5 WCET computation under multiple bus contexts

7.5.1 Execution context of a basic block

Computing bus context without loops In the previous section, we have discussed the pipeline modeling of a basic block B in isolation. However, to correctly compute the execution time of B, we need to consider 1) contentions (for functional units) and data dependencies among instructions prior to B and instructions in B; 2) contentions among instructions after B and instructions in B. Set of instructions before (after) B which directly affect the execution time of B is called the *prologue* (*epilogue*) of B [17]. B may have multiple prologues and epilogues

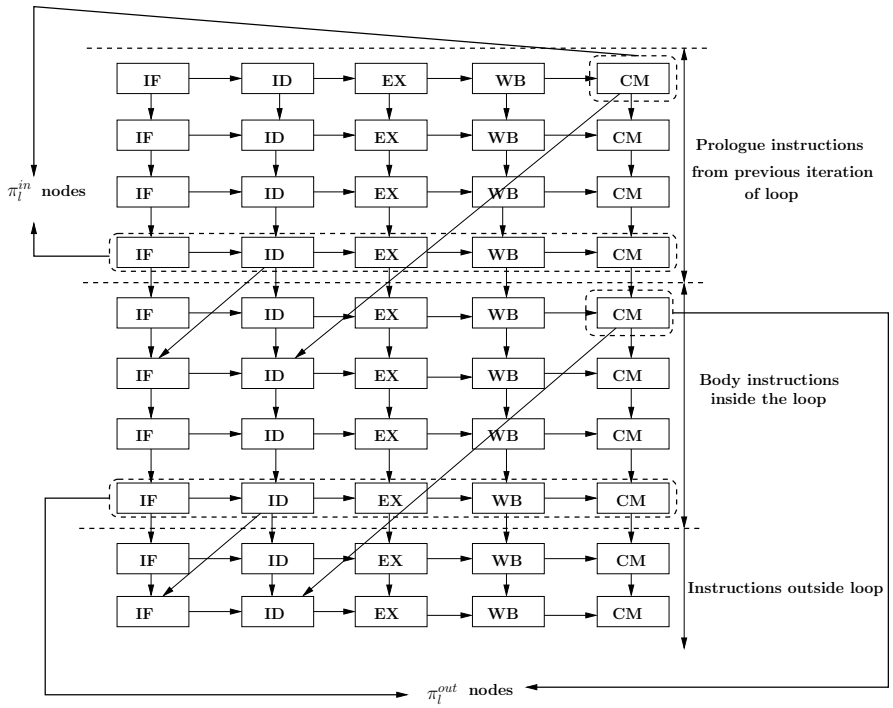


Figure 7.3: π_l^{in} and π_l^{out} nodes shown with the example of a sample execution graph. π_l^{in} nodes propagate bus contexts across iterations, whereas, π_l^{out} nodes propagate bus contexts outside of loop.

due to the presence of multiple program paths. However, the size of any prologue or epilogue is bounded by the total size of IFQ and ROB. To distinguish the execution contexts of a basic block B, execution graphs are constructed for each possible combination of prologues and epilogues of B. Each execution graph of B contains the instructions from B itself (called *body*) and the instructions from one possible prologue and epilogue. Assume we compute the incoming (outgoing) bus context $O_i^{in}(p,e)$ ($O_i^{out}(p,e)$) at body node i for prologue p and epilogue e (using the technique described in Section 7.4). After we finish the analysis of B for all possible combinations of prologues and epilogues, we compute an *over-approximation* of O_i^{in} (O_i^{out}) by *merge* operation as follows:

$$O_i^{in} = \bigcup_{p,e} O_i^{in}(p,e) \tag{7.10}$$

$$O_i^{out} = \bigcup_{p,e} O_i^{out}(p,e) \tag{7.11}$$

Clearly, O_i^{in} (O_i^{out}) captures an over-approximation of the bus context at the entry (exit) of node i, irrespective of any prologue or epilogue of B.

Computing bus context in the presence of loops In the presence of loops, a basic block can be executed with different bus contexts at different iterations of the loop. The bus contexts at different iterations depend on the set of instructions which can propagate TDMA offsets across loop iterations. For each loop l, we compute two sets of nodes — π_l^{in} and π_l^{out}. π_l^{in} are the set of pipeline stages which can propagate TDMA offsets across iterations, whereas, π_l^{out} are the set of pipeline stages which could propagate TDMA offsets outside of the loop. Therefore, π_l^{in} corresponds to the pipeline stages of instructions inside l which resolve *loop carried dependency* (due to resource constraints, pipeline structural constraints or true data dependency). On the other hand, π_l^{out} corresponds to the pipeline stages of instructions inside l which resolve the dependency of instructions outside of l. Figure 7.3 demonstrates the π_l^{out} and π_l^{in} nodes for a sample execution graph. The bus context at the entry of all *non-first* loop iterations can be captured as $(O_{x1}^{in}, O_{x2}^{in}, \ldots, O_{xn}^{in})$ where $\pi_l^{in} = \{x1, x2, \ldots, xn\}$. The bus context at the first iteration is computed from the bus contexts of instructions prior to l (using the technique described in Section 7.4). Finally, O_{xi}^{out} for any $xi \in \pi_l^{out}$ can be responsible for affecting the execution time of any basic block outside of l.

7.5.2 Bounding the execution count of a bus context

Foundation As discussed in the preceding, a basic block inside some loop may execute under different bus contexts. For all *non-first* iterations, a loop l is entered with bus context $(O_{x1}^{in}, O_{x2}^{in}, \ldots, O_{xn}^{in})$ where $\{x1, x2, \ldots, xn\}$ are the set of π_l^{in} nodes as described in Figure 7.3. These bus contexts are computed during an iterative analysis of the loop l (described below). On the other hand, the bus context at the first iteration of l is a tuple of *TDMA offsets* propagated from outside of l to some pipeline stage inside l. Note that the bus context at the first iteration of l is computed by following the general procedure as described in Section 7.4.

In this section, we shall show how the execution count of different bus contexts can be bounded by generating additional ILP constraints. These additional constraints are added to a global ILP formulation to find the WCET of the entire program. We begin with the following notations:

Ω_l The set of all bus contexts that may reach loop l in any iteration.

Ω_l^s The set of all bus contexts that may reach loop l at first iteration. Clearly, $\Omega_l^s \subseteq \Omega_l$. Moreover, if l is contained inside some outer loop, l would be invoked more than once. As a result, Ω_l^s may contain more than one element. Note that Ω_l^s can be computed as a tuple of TDMA offsets propagated from outside of l to some pipeline stage inside l. Therefore, Ω_l^s can be computed during the procedure described in Section 7.4. If l is an inner loop, an element of Ω_l^s is computed (as described in Section 7.4) for each analysis invocation of the loop immediately enclosing l.

G_l^s For each $s_0 \in \Omega_l^s$, we build a *flow graph* $G_l^s = (V_l^s, F_l^s)$ where $V_l^s \subseteq \Omega_l$. The graph G_l^s captures the transitions among different bus contexts across loop iterations. An edge $f_{w_1 \rightarrow w_2} = (w_1, w_2) \in F_l^s$ exists (where $w_1, w_2 \in \Omega_l$) if and only if l can be entered with bus context w_1 at some iteration n and with bus context w_2 at iteration $n+1$. Note that G_l^s cannot be infinite, as we have only finitely few bus contexts that are the nodes of G_l^s.

M_l^w Number of times the body of loop l is entered with bus context $w \in \Omega_l$ in any iteration.

$M_l^{w_1 \rightarrow w_2}$ Number of times l can be entered with bus context w_1 at some iteration n and with bus context w_2 at iteration $n+1$ (where $w_1, w_2 \in \Omega_l$). Clearly, if $f_{w_1 \rightarrow w_2} \notin F_l^s$ for any flow graph G_l^s, $M_l^{w_1 \rightarrow w_2} = 0$.

Construction of G_l^s For each loop l and for each $s_0 \in \Omega_l^s$, we construct a flow graph G_l^s. Initially, G_l^s contains a single node representing bus context $s_0 \in \Omega_l^s$. After analyzing all the basic blocks inside l (using the technique described in Section 7.4), we may get a new bus context at some node $i \in \pi_l^{in}$ (recall that π_l^{in} are the set of execution graph nodes that may propagate bus context across loop iterations). As a byproduct of this process, we also get the WCET of all basic blocks inside l when the body of l is entered with bus context s_0. Let us assume that for any $s \in \Omega_l \setminus \Omega_l^s$ and $i \in \pi_l^{in}$, $s(i)$ represents the bus context O_i^{in}. Suppose we get a new bus context $s_1 \in \Omega_l$ after analyzing the body of l once. Therefore, we add an edge from s_0 to s_1 in G_l^s. We continue expanding G_l^s until $s_n(i) \subseteq s_k(i)$ for all $i \in \pi_l^{in}$ and for some $1 \leq k \leq n-1$ (where $s_n \in \Omega_l$ represents the bus context at the entry of l after it is analyzed n times). In this case, we finish the construction of G_l^s by adding a backedge from s_{n-1} to s_k. We also stop expanding G_l^s if we have expanded as many times as the relative loop bound of l. Note that G_l^s contains at least two nodes, as the bus context at first loop iteration is always

distinguished from the bus contexts in any other loop iteration.

It is worth mentioning that the construction of G_l^s is *much less computationally intensive* than a full unrolling of l. The bus context at the entry of l quickly reaches a fixed-point and we can stop expanding G_l^s. In our experiments, we found that the number of nodes in G_l^s never exceeds ten. For very small loop bounds (typically less than 5), the construction of G_l^s continues till the loop bound. For larger loop bounds, most of the time, the construction of G_l^s reaches the diverged bus context $[0, \ldots, S_l\mathcal{C} - 1]$ quickly (in less than ten iterations). As a result, through a small node count in G_l^s, we are able to avoid the computationally intensive unrolling of every loop.

Generating separate ILP constraints Using each flow graph G_l^s for loop l, we generate ILP constraints to distinguish different bus contexts under which a basic block can be executed. In an abuse of notation, we shall use $w.i$ to denote that the basic block i is reached with bus context $w.i$ when the immediately enclosing loop of i is reached with bus context w in any iteration. The following ILP constraints are generated to bound the value of M_l^w:

$$\forall w \in \Omega_l : \sum_{x \in \Omega_l} M_l^{x \to w} = M_l^w \tag{7.12}$$

$$\forall w \in \Omega_l : \; M_l^w - 1 \leq \sum_{x \in \Omega_l} M_l^{w \to x} \leq M_l^w \tag{7.13}$$

$$\sum_{w \in \Omega_l} M_l^w = N_{l.h} \tag{7.14}$$

where $N_{l.h}$ denotes the number of times the header of loop l is executed. Equations 7.12-7.13 generate standard flow constraints from each graph G_l^s, constructed for loop l. Special constraints need to be added for the bus contexts with which the loop is entered at the first iteration and at the last iteration. If w is a bus context with which loop l is entered at the last iteration, M_l^w is more than the execution count of outgoing flows (*i.e.* $M_l^{w \to x}$). Equation 7.13 takes this special case into consideration. On the other hand, Equation 7.14 bounds the aggregate execution count of all possible contexts $w \in \Omega_l$ with the total execution count of the loop header. Note that $N_{l.h}$ will further be involved in defining the CFG structural constraints, which relate the execution count of a basic block with the execution count of its incoming and

outgoing edges [9]. Equations 7.12-7.14 do not ensure that whenever loop l is invoked, the loop must be executed at least once with some bus context in Ω_l^s. We add the following ILP constraints to ensure this:

$$\forall w \in \Omega_l^s : \; M_l^w \geq N_{l.h}^{w.h} \tag{7.15}$$

Here $N_{l.h}^{w.h}$ denotes the number of times the header of loop l is executed with bus context w. The value of $N_{l.h}^{w.h}$ is further bounded by the CFG structural constraints.

The constraints generated by Equations 7.12-7.15 are sufficient to derive the WCET of a basic block in the presence of non-nested loops. In the presence of nested loops, however, we need additional ILP constraints to relate the bus contexts at different loop nests. Assume that the loop l is enclosed by an outer loop l'. For each $w' \in \Omega_{l'}$, we may get a different element $s_0 \in \Omega_l^s$ and consequently, a different $G_l^s = (V_l^s, E_l^s)$ for loop l. Therefore, we have the following ILP constraints for each flow graph G_l^s:

$$\forall G_l^s = (V_l^s, E_l^s) : \sum_{w \in V_l^s} M_l^w \leq bound_l * (\sum_{w' \in parent(G_l^s)} M_{l'}^{w'}) \tag{7.16}$$

where $bound_l$ represents the *relative loop bound* of l and $parent(G_l^s)$ denotes the set of bus contexts in $\Omega_{l'}$ for which the flow graph G_l^s is constructed at loop l. The left-hand side of Equation 7.16 accumulates the execution count of all bus contexts in the flow graph G_l^s. The total execution count of all bus contexts in V_l^s is bounded by $bound_l$, for each construction of G_l^s (as $bound_l$ is the relative loop bound of l). Since G_l^s is constructed $\sum_{w' \in parent(G_l^s)} M_{l'}^{w'}$ times, the total execution count of all bus contexts in V_l^s is bounded by the right hand side of Equation 7.16.

Finally, we need to bound the execution count of any basic block i (immediately enclosed by loop l), with different bus contexts. We generate the following two constraints to bound this value:

$$\sum_{w \in \Omega_l} N_i^{w.i} = N_i \tag{7.17}$$

$$\forall w \in \Omega_l : \; N_i^{w.i} \leq M_l^w \tag{7.18}$$

where N_i represents the total execution count of basic block i and $N_i^{w.i}$ represents the execution count of basic block i with bus context $w.i$. Equation 7.18 tells the fact that basic block i can execute with bus context $w.i$ at some iteration of l only if l is reached with bus context w at the

same iteration (by definition). N_i will be further constrained through the structure of program's CFG, which we exclude in our discussion.

Computing bus contexts at loop exit To derive the WCET of the whole program, we need to estimate the bus context exiting a loop l (say O_l^{exit}). A recently proposed work ([41]) has shown the computation of O_l^{exit} without a full loop unrolling. In this paper, we use a similar technique as in [41] with one important difference: In [41], a single offset graph G_{off} is maintained, which tracks the outgoing bus context from each loop iteration. Once G_{off} got stabilized, a separate ILP formulation on G_{off} derives the value of O_l^{exit}. In the presence of pipelined architectures, O_i^{out} for any $i \in \pi_l^{out}$ could be responsible for propagating bus context outside of l (refer to Figure 7.3). Therefore, a separate offset graph is maintained for each $i \in \pi_l^{out}$ (say G_{off}^i) and an ILP formulation for each G_{off}^i can derive an estimation of the bus context exiting the loop (say O_i^{exit}). In [41], it has been proved that the computation of O_l^{exit} is always an *over-approximation* (*i.e. sound*). Given that the value of each O_i^{out} is *sound*, it is now straightforward to see that the computation of each O_i^{exit} is also *sound*. For details of this analysis, readers are further referred to [41].

7.6 Effect of branch prediction

Presence of branch prediction introduces additional complexity in WCET computation. If a conditional branch is mispredicted, the timing of the mispredicted instructions need to be computed. Mispredicted instructions introduce additional conflicts in L1 and L2 cache which need to be modeled for a sound WCET computation. Similarly, branch misprediction will also affect the bus delay suffered by the subsequent instructions. In the following, we shall describe how our framework models the interaction of branch predictor on cache and bus. We assume that there could be at most one unresolved branch at a time. Therefore, the number of mispredicted instructions is bounded by the number of instructions till the next branch as well as the total size of instruction fetch queue and reorder buffer.

7.6.1 Effect on cache for speculative execution

Abstract-interpretation-based cache analysis produces a fixed point on abstract cache content at the entry (denoted as ACS_i^{in}) and at the exit (denoted as ACS_i^{out}) of each basic block i. If a basic block i has multiple predecessors, output cache states of the predecessors are *joined* to

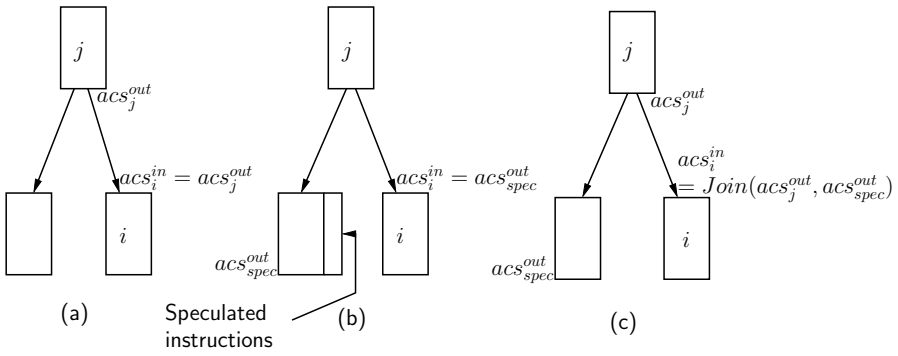


Figure 7.4: (a) Computation of acs_i^{in} when the edge $j \rightarrow i$ is correctly predicted, (b) Computation of acs_i^{in} when the edge $j \rightarrow i$ is mispredicted, (c) A *safe* approximation of acs_i^{in} by considering both correct and incorrect prediction of edge $j \rightarrow i$.

produce the input cache state of basic block i. Consider an edge $j \rightarrow i$ in the program's CFG. If $j \rightarrow i$ is an unconditional edge, computation of ACS_i^{in} does not require any change. However, if $j \rightarrow i$ is a conditional edge, the condition could be *correctly* or *incorrectly* predicted during the execution. For a correct prediction, the cache state ACS_i^{in} is still *sound*. On the other hand, for incorrect prediction, ACS_i^{in} must be updated with the memory blocks accessed at the mispredicted path. We assume that there could be at most one unresolved branch at a time. Therefore, the number of mispredicted instructions is bounded by the number of instructions till the next branch as well as the total size of instruction fetch queue and reorder buffer. To maintain a *safe* cache state at the entry of each basic block i, we join the two cache states arising due to the *correct* and *incorrect* predictions of conditional edge $j \rightarrow i$. We demonstrate the entire scenario through an example in Figure 7.4. In Figure 7.4, we demonstrate the procedure for computing the abstract cache state at the entry of a basic block i. Basic block i is conditionally reached from basic block j. To compute a *safe* cache content at the entry of basic block i, we combine two different possibilities —- one when the respective branch is *correctly* predicted (Figure 7.4(a)) and the other when the respective branch is *incorrectly* predicted (Figure 7.4(b)). The combination is performed through an abstract *join* operation, which depends on the type of analysis (*must* or *may*) being computed. A stabilization on the abstract cache contents at the entry and exit of each basic block is achieved through conventional fixed point analysis.

7.6.2 Effect on bus for speculative execution

Due to branch misprediction, some additional instructions might be fetched from the mispredicted path. As described in Section 7.5, an execution graph for each basic block B contains

a *prologue* (instructions before B which directly affect the execution time of B). If the last instruction of the prologue is a conditional branch, the respective execution graph is augmented with the instructions along the mispredicted path ([17]). Since the propagation of *bus context* is entirely performed on the execution graph (as shown in Section 7.4), our shared bus analysis remains unchanged, except the fact that it works on an augmented execution graph (which contains instructions from the mispredicted path) in the presence of speculative execution.

7.6.3 Computing the number of mispredicted branches

In the presence of a branch predictor, each conditional edge $j \rightarrow i$ in the program CFG can be correctly or incorrectly predicted. Let us assume $E_{j\rightarrow i}$ denotes the total number of times control flow edge $j \rightarrow i$ is executed and $E^c_{j\rightarrow i}$ ($E^m_{j\rightarrow i}$) denotes the number of times the control flow edge $j \rightarrow i$ is executed due to correct (incorrect) branch prediction. Clearly, $E_{j\rightarrow i} = E^c_{j\rightarrow i} + E^m_{j\rightarrow i}$. Value of $E_{j\rightarrow i}$ is further bounded by CFG structural constraints. On the other hand, values of $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ depend on the type of branch predictor. We use our prior work ([18]), where we have shown how to bound the values of $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ for history based branch predictors. The constraints generated on $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ are as well captured in the global ILP formulation to compute the whole program WCET. We exclude the details of branch predictor modeling in this paper — interested readers are referred to [18].

7.7 WCET computation of an entire program

We compute the WCET of the entire program with N basic blocks by using the following objective function:

$$Maximize\, T = \sum_{i=1}^{N} \sum_{j\rightarrow i} \sum_{w\in\Omega_i} t^{c,w}_{j\rightarrow i} * E^{c,w}_{j\rightarrow i} + t^{m,w}_{j\rightarrow i} * E^{m,w}_{j\rightarrow i} \tag{7.19}$$

Ω_i denotes the set of all bus contexts under which basic block i can execute. Basic block i can be executed with different bus contexts. However, the number of elements in Ω_i is always bounded by the number of bus contexts entering the loop immediately enclosing i (refer to Section 7.5). $t^{c,w}_{j\rightarrow i}$ denotes the WCET of basic block i when the basic block i is reached from basic block j, the control flow edge $j \rightarrow i$ is correctly predicted and i is reached with bus context $w \in \Omega_i$. Similarly, $t^{m,w}_{j\rightarrow i}$ denotes the WCET of basic block i under the same bus context but when the

control flow edge $j \rightarrow i$ was mispredicted. Note that both $t^{c,w}_{j\rightarrow i}$ and $t^{m,w}_{j\rightarrow i}$ are computed during the iterative pipeline modeling (with the modifications proposed in Section 7.4). $E^{c,w}_{j\rightarrow i}$ ($E^{m,w}_{j\rightarrow i}$) denotes the number of times basic block i is reached from basic block j with bus context w and when the control flow edge $j \rightarrow i$ is correctly (incorrectly) predicted. Therefore, we have the following two constraints:

$$E^{c}_{j\rightarrow i} = \sum_{w\in\Omega_i} E^{c,w}_{j\rightarrow i}, \quad E^{m}_{j\rightarrow i} = \sum_{w\in\Omega_i} E^{m,w}_{j\rightarrow i} \tag{7.20}$$

Constraints on $E^{c}_{j\rightarrow i}$ and $E^{m}_{j\rightarrow i}$ are proposed by the ILP-based formulation in [18]. On the other hand, $E^{c,w}_{j\rightarrow i}$ and $E^{m,w}_{j\rightarrow i}$ are bounded by the CFG structural constraints ([9]) and the constraints proposed by Equations 7.12-7.18 in Section 7.5. Note that in Equations 7.12-7.18, we only discuss the ILP constraints related to the bus contexts. Other ILP constraints, such as CFG structural constraints and user constraints, are used in our framework for an IPET implementation.

Finally, the WCET of the program maximizes the objective function in Equation 7.19. Any ILP solver (*e.g.* CPLEX) can be used for the same purpose.

7.8 Soundness and termination of analysis

In this section, we shall first provide the basic ideas for the proof of the *soundness* of our analysis framework and subsequently, elaborate each point.

7.8.1 Overall idea about soundness

The heart of *soundness* guarantee follows from the fact that we represent the timing of each pipeline stage as an *interval*. Recall that the active timing interval of each pipeline stage is captured by $INTV_i = [earliest[t_i^{ready}], latest[t_i^{finish}]]$. Therefore, as long as we can guarantee that $INTV_i$ is always an *over-approximation* of the actual timing interval of the corresponding pipeline stage in any concrete execution, we can also guarantee the *soundness* of our analysis. To ensure that the interval $INTV_i$ is always an over-approximation, we have to consider all possible latencies suffered by any pipeline stage. The latency of a pipeline stage, on the other hand, may be influenced by the following factors:

Cache miss penalty Only NC categorized memory references may have variable latencies. Our analysis represents this variable latency as an interval $[lo, hi]$ (Equation 7.1) where lo (hi) represents the latency of a cache hit (miss).

Functional unit latency Some functional units may have variable latencies depending on operands (*e.g.* multiplier unit). For such functional units, we consider the EX pipeline stage latency as an interval $[lo, hi]$ where lo (hi) represents the minimum (maximum) possible latency of the corresponding functional unit.

Contention to access functional units A pair of instructions may delay each other by contending for the same functional unit. Since only EX stage may suffer from contention, two different instructions may contend for the same functional unit only if the timing intervals of respective EX stages *overlap*. For any pipeline stage i, an upper bound on contention (say $CONT_i^{max}$) is computed by accounting the cumulative effect of contentions created by all the *overlapping* pipeline stages (which access the same functional unit as i). We do not compute a lower bound on contention and conservatively assume a safe lower bound of 0. Finally, we add $[0, CONT_i^{max}]$ with the timing interval of pipeline stage i. Clearly, $[0, CONT_i^{max}]$ covers all possible latencies suffered by pipeline stage i due to contention.

Bus access delay Bus access delay of a pipeline stage depends on incoming bus contexts (O_i^{in}). Computation of O_i^{in} is always an over-approximation as evidenced by Equation 7.7 and Equation 7.9. Therefore, we can always compute the interval spanning from *minimum* to *maximum* bus delay using O_i^{in} (Equation 7.4 and Equation 7.5).

In the following description, we shall argue how our analysis maintain *soundness* for each of these four scenarios.

7.8.2 Detailed proofs

Property 7.8.1. Functional unit latency considered during analysis is always *sound*. More precisely, any functional unit latency that may appear in a concrete execution, is considered during WCET analysis.

Proof. If a functional unit has fixed latency, the soundness follows *trivially*. However, a functional unit may have variable latency (*e.g.* multiplier unit). Assume lo (hi) represents the

minimum (maximum) latency that could possibly be suffered by using functional unit f. Our WCET analysis uses an interval $[lo, hi]$ to represent the execution latency (*i.e.* the latency of EX stage in the pipeline) for all the instructions which may use f. In this way, we are able to handle the worst case which may arise due to a lower functional unit latency. □

Property 7.8.2. Cache access latencies considered during analysis is always sound. Therefore, WCET analysis considers all possible cache access latencies which may appear in a concrete execution.

Proof. Recall that memory references are classified as all-hit (AH), all-miss (AM) and unclassified (NC) in L1 and (shared) L2 cache. The *soundness* of categorizing a memory reference either AH or AM in L1 or (shared) L2 cache follows from the soundness of analyses proposed in [9] and [15]. On the other hand, the soundness of our analysis directly follows from Equation 7.1. Note that the latency considered for NC categorized memory reference (Equation 7.1) captures the entire interval — ranging from cache hit latency to cache miss latency. Therefore, our analysis can handle the worst case which may arise due to a *cache hit* (instead of a *cache miss*) for a particular memory reference. □

We propose the following properties which are essential for understanding the *soundness* of shared bus analysis.

Property 7.8.3. Consider an execution graph of a basic block B and assume $INIT_B$ represents the set of execution graph nodes without any predecessor. Assume two different execution contexts of basic block B say c_1 and c_2. Further assume $O_j^{in}(c_1)$ ($O_j^{in}(c_2)$) and $O_j^{out}(c_1)$ ($O_j^{out}(c_2)$) represent the incoming and outgoing bus context, respectively, at any execution graph node j with execution context c_1 (c_2). Finally assume that each EX stage in the execution context c_2 experiences at least as much contention as in the execution context c_1. For any execution graph node j, the following property holds: if $O_j^{in}(c_1) \not\subseteq O_j^{in}(c_2)$, then $O_i^{in}(c_1) \not\subseteq O_i^{in}(c_2)$ for at least one $i \in INIT_B$.

Proof. For $j \in INIT_B$, our claim trivially follows. Therefore, assume $j \notin INIT_B$. We prove our claim by contradiction. We assume that $O_i^{in}(c_1) \subseteq O_i^{in}(c_2)$ for all $i \in INIT_B$, but

$O_j^{in}(c_1) \nsubseteq O_j^{in}(c_2)$. Note that any execution graph is *acyclic* and consequently, it has a valid *topological ordering*. We prove that the contradiction is invalid (*i.e.* $O_j^{in}(c_1) \subseteq O_j^{in}(c_2)$) by induction on the topological order n of execution graph nodes.

Base case $n = 1$. These are the nodes in $INIT_B$. Therefore, the claim directly follows from our assumption.

Induction step Assume all nodes in the execution graph which have topological order $\leq k$ validates our claim. We prove that any node j having topological order $\geq k + 1$ validates our claim as well. If we assume a contradiction then $O_j^{in}(c_1) \nsubseteq O_j^{in}(c_2)$. However, it is only possible if one of the following conditions hold for some predecessor p' of j (refer to Equation 7.9):

- $earliest[t_{p'}^{finish}](c_1) < earliest[t_{p'}^{finish}](c_2)$ or
- $latest[t_{p'}^{finish}](c_1) > latest[t_{p'}^{finish}](c_2)$ or
- $O_{p'}^{out}(c_1) \nsubseteq O_{p'}^{out}(c_2)$.

where $earliest[t_i^{finish}](c_1)$ ($latest[t_i^{finish}](c_1)$) and $earliest[t_i^{finish}](c_2)$ ($latest[t_i^{finish}](c_2)$) represent the *earliest* (*latest*) finish time of node i in the execution contexts c_1 and c_2, respectively. As any EX stage in the execution context c_2 experiences more contention than in the execution context c_1 (our assumption), any of the above three conditions can hold only if $O_{p'}^{in}(c_1) \nsubseteq O_{p'}^{in}(c_2)$. Following the same argument and going backward in the topological order of the execution graph, we must have a predecessor p_0 which has topological order $\leq k$ and $O_{p_0}^{in}(c_1) \nsubseteq O_{p_0}^{in}(c_2)$. This contradicts our induction hypothesis. Therefore, our initial claim was invalid.

This property ensures that the bus contexts reaching at basic block B can precisely be encoded by the set of bus contexts reaching at $INIT_B$, ignoring functional unit contentions (since the bus context at any node in the execution graph can grow only if the bus context at some node $i \in INIT_B$ grows). The following property ensures that the same is true even in the presence of functional unit contentions. □

Property 7.8.4. Consider an execution graph of a basic block B and assume $INIT_B$ represents the set of execution graph nodes without any predecessor. Assume two different execution

contexts of basic block B say c_1 and c_2. Further assume $O_j^{in}(c_1, n)$ ($O_j^{in}(c_2, n)$) and $O_j^{out}(c_1, n)$ ($O_j^{out}(c_2, n)$) represent the incoming and outgoing bus context, respectively, at any execution graph node j with execution context c_1 (c_2) and at the n-th iteration of pipeline modeling. Finally assume $CR_n(c_1)$ ($CR_n(c_2)$) represents the contention relation in the execution context c_1 (c_2) and at the n-th iteration of pipeline modeling. For any execution graph node j, the following property holds: if $O_i^{in}(c_1, n) \subseteq O_i^{in}(c_2, n)$ for all $i \in INIT_B$ then $O_j^{in}(c_1, n) \subseteq O_j^{in}(c_2, n)$ for any execution graph node j and $CR_n(c_1) \subseteq CR_n(c_2)$ over different iterations n of pipeline modeling.

Proof. Assume $earliest[t_i^{ready}, n](c_1)$ ($earliest[t_i^{ready}, n](c_2)$) represents the earliest ready time of execution graph node i in the execution context c_1 (c_2) and at n-th iteration of pipeline modeling. Similarly, $latest[t_i^{finish}, n](c_1)$ ($latest[t_i^{finish}, n](c_2)$) represents the latest finish time of execution graph node i in the execution context c_1 (c_2) and at n-th iteration of pipeline modeling. We prove our claim by an induction on the number of iterations (n) of pipeline modeling.

Base case $n = 1$. We start with all possible pairs of instructions in the contention relation (*i.e.* we assume that every pair of instructions which may use same functional unit, can potentially delay each other). Therefore, $CR_1(c_1) = CR_1(c_2)$. Property 7.8.3 ensures that $O_j^{in}(c_1, 1) \subseteq O_j^{in}(c_2, 1)$ for any execution graph node j. Consequently, for any execution graph node j, we can conclude that

- $earliest[t_j^{ready}, 1](c_1) \geq earliest[t_j^{ready}, 1](c_2)$
- $latest[t_j^{finish}, 1](c_1) \leq latest[t_j^{finish}, 1](c_2)$

Therefore, $CR_2(c_1) \subseteq CR_2(c_2)$ as the timing interval of any execution graph node is *coarser* in the execution context c_2 compared to the corresponding timing interval in the execution context c_1.

Induction step We assume that $CR_n(c_1) \subseteq CR_n(c_2)$ and $O_j^{in}(c_1, n) \subseteq O_j^{in}(c_2, n)$ for any execution graph node j. We shall prove that $CR_{n+1}(c_1) \subseteq CR_{n+1}(c_2)$ and $O_j^{in}(c_1, n+1) \subseteq O_j^{in}(c_2, n+1)$ for any execution graph node j. We shall prove the same by contradiction (*i.e.* assume that $CR_{n+1}(c_1) \nsubseteq CR_{n+1}(c_2)$). Informally, we have at least two execution graph

nodes i and j which have *disjoint* timing intervals in the execution context c_1 but have *overlapping* timing intervals in the execution context c_2. This is only possible if one of the following conditions hold:

- $earliest[t_i^{ready}, n+1](c_1) < earliest[t_i^{ready}, n+1](c_2)$
- $earliest[t_j^{ready}, n+1](c_1) < earliest[t_j^{ready}, n+1](c_2)$.
- $latest[t_i^{finish}, n+1](c_1) > latest[t_i^{finish}, n+1](c_2)$
- $latest[t_j^{finish}, n+1](c_1) > latest[t_j^{finish}, n+1](c_2)$

However, above situation may arise only if one of the following two conditions holds:

- $O_k^{in}(c_1, n+1) \nsubseteq O_k^{in}(c_2, n+1)$ for some execution graph node k. Since $CR_n(c_1) \subseteq CR_n(c_2)$, Property 7.8.3 ensures $O_p^{in}(c_1, n+1) \nsubseteq O_p^{in}(c_2, n+1)$ for at least one node p which does not have any predecessor. This is a contradiction as $O_p^{in}(c_1, n+1) = O_p^{in}(c_1, n)$ and $O_p^{in}(c_2, n+1) = O_p^{in}(c_2, n)$ and therefore, $O_p^{in}(c_1, n+1) = O_p^{in}(c_1, n) \subseteq O_p^{in}(c_2, n) = O_p^{in}(c_2, n+1)$.
- $CR_n(c_1) \nsubseteq CR_n(c_2)$, which may increase $latest[t_i^{finish}, n+1](c_1)$ with respect to the value of $latest[t_i^{finish}, n+1](c_2)$ for some node i. However, this is a contradiction of our induction hypothesis.

This property generalizes the previous Property 7.8.3 by considering functional unit contentions.

□

Property 7.8.5. Consider an execution graph of a basic block B and assume $INIT_B$ represents the set of execution graph nodes without any predecessor. Assume two different execution contexts of basic block B say c_1 and c_2. Further assume $O_j^{in}(c_1)$ ($O_j^{in}(c_2)$) and $O_j^{out}(c_2)$ ($O_j^{out}(c_2)$) represent the incoming and outgoing bus context, respectively, at any execution graph node j with execution context c_1 (c_2). If $O_i^{in}(c_1) \subseteq O_i^{in}(c_2)$ for all $i \in INIT_B$, WCET of basic block B in the execution context c_2 is always at least equal to the WCET of basic block B in the execution context c_1.

Proof. This claim follows directly from Properties 7.8.3-7.8.4. If $O_i^{in}(c_1) \subseteq O_i^{in}(c_2)$ for all nodes $i \in INIT_B$, then according to Properties 7.8.3-7.8.4, $O_j^{in}(c_1) \subseteq O_j^{in}(c_2)$ for any execution graph node j. Since the bus context at any execution graph node with the execution context c_2 subsumes the respective bus contexts with the execution context c_1, we can conclude that the WCET of basic block B with the execution context c_2 is at least equal to the WCET of basic block B with the execution context c_1. □

Property 7.8.6. Consider any non-nested loop l. Assume $O_i^{in}(m)$ represents the incoming bus context of any execution graph node i at m-th iteration of loop. Consider two different iterations m' and m'' of loop l. If $O_{xi}^{in}(m') \subseteq O_{xi}^{in}(m'')$ for all $xi \in \pi_l^{in}$, $O_{xi}^{in}(m'+1) \subseteq O_{xi}^{in}(m''+1)$ for all $xi \in \pi_l^{in}$. Moreover, WCET of any basic block inside loop l at iteration m'' must be at least equal to the WCET of the corresponding basic block at iteration m'.

Proof. By definition, π_l^{in} corresponds to the set of pipeline stages which resolve *loop carried dependency* (either due to resource constraints, pipeline structural constraints or true data dependency). This direct dependency is specified through directed edges in the execution graph (as shown in Figure 7.3). We first prove that $O_j^{in}(m') \subseteq O_j^{in}(m'')$ for any execution graph node j that corresponds to some instruction inside l. We prove our claim by induction on the topological order n of basic blocks in l.

Base case $n = 1$. This is the loop header H. By using an exactly similar proof as in properties 7.8.3-7.8.4, we can show that if $O_{xi}^{in}(m') \subseteq O_{xi}^{in}(m'')$ for all $xi \in \pi_l^{in}$, $O_i^{in}(m') \subseteq O_i^{in}(m'')$ for any node i in the execution graph of H.

Induction step Assume our claim holds for all basic blocks having topological order $\leq k$. We shall prove that our claim holds for all basic blocks having topological order $\geq k+1$. However, using our methodology for proving Properties 7.8.3-7.8.4, we can easily show that if the bus context for some basic block (having topological order $\geq k+1$) at iteration m'' is not an *over-approximation* of the bus context of the same basic block at iteration m', it could be either of two following reasons:

- The bus context at iteration m'' is not an *over-approximation* of the bus context at iteration m' for some basic block having topological order $\leq k$, contradicting our induction hypothesis;

- For some $xi \in \pi_l^{in}$, $O_{xi}^{in}(m') \not\subseteq O_{xi}^{in}(m'')$, contradicting our assumption.

Since the bus contexts computed at each basic block at iteration m'' subsume the corresponding bus contexts at iteration m', $O_{xi}^{in}(m'+1) \subseteq O_{xi}^{in}(m''+1)$ for all $xi \in \pi_l^{in}$. For the same reason, WCET of any basic block inside l at m''-th iteration is at least equal to the WCET of the corresponding basic block at iteration m'.

Recall that to track the bus contexts at different loop iterations, we construct a flow graph G_l^s. We terminate the construction of G_l^s after k ($k \geq 1$) iterations only if for all $i \in \pi_l^{in}$, $O_i^{in}(k) \subseteq O_i^{in}(j)$ where $1 \leq j < k$. We add a backedge from $k-1$-th bus context to j-th bus context to terminate the construction of G_l^s. The bus context at some loop iteration n is computed from G_l^s by following a path of length n from the initial node. In case n is less than the number of nodes in G_l^s, it is straightforward to see that the computed bus context is always an over-approximation (as evidenced by Equation 7.7 and Equation 7.9). In case n is more than the number of nodes in G_l^s (*i.e.* backedge in G_l^s is followed at least once to compute the bus context), the above property ensures that the bus context computed by the flow graph is always an over-approximation.

In the following property, we shall generalize the result for any loop (nested or non-nested).

□

Property 7.8.7. Consider any loop l. Assume $O_i^{in}(m)$ represents the incoming bus context of any execution graph node i at m-th iteration of loop. Consider two different iterations m' and m'' of loop l. If $O_{xi}^{in}(m') \subseteq O_{xi}^{in}(m'')$ for all $xi \in \pi_l^{in}$, $O_{xi}^{in}(m'+1) \subseteq O_{xi}^{in}(m''+1)$ for all $xi \in \pi_l^{in}$. Moreover, if l contains some loop l', O_{xj}^{exit} computed at m''-th iteration of l always over-approximates O_{xj}^{exit} computed at m'-th iteration of l, for every $xj \in \pi_{l'}^{out}$.

Proof. Let us first consider some loop l which contains only non-nested loops. Let us assume a topological order of all inner loops inside l and assume l^x represents the inner loop contained in l, which is preceded by $x-1$ other inner loops inside l, in topological order. We first prove

that $O_j^{in}(m') \subseteq O_j^{in}(m'')$ for any execution graph node j that corresponds to some instruction inside l. We also prove that for any inner loop l^x and for all $j \in \pi_{l^x}^{out}$, O_j^{exit} computed at m''-th iteration of l is always an over-approximation of O_j^{exit} computed at m'-th iteration of l.

For any basic block i inside l, assume that n_i is the number of loop exit edges appearing prior in topological order of i. We assume that each loop has a single exit node. If some loop has multiple exits, we can assume an empty node which post-dominates all the exit nodes of the loop. We prove our claim by induction on n_i.

Base case $n_i = 0$. Therefore, we have the two following possibilities:

- (Case I) i is a basic block which is immediately enclosed by loop l.
- (Case II) i is a basic block which is immediately enclosed by loop l^1 and l^1 is the *first loop* contained inside l, following a topological order.

For Case I, Property 7.8.6 ensures that $O_j^{in}(m') \subseteq O_j^{in}(m'')$ for all nodes j that corresponds to the instructions in basic block i.

For Case II, basic block i may have different bus contexts at different iterations of loop l^1. We shall prove that the bus context computed for basic block i at any iteration of l^1 validates our claim. Assume $O_j^{in}(x, x')$ ($O_j^{out}(x, x')$) represents the incoming (outgoing) bus context at the execution graph node j at x-th iteration of l and at x'-th iteration of l^1. Properties 7.8.3-7.8.4 ensure that $O_{xj}^{in}(m', 1) \subseteq O_{xj}^{in}(m'', 1)$ for all $xj \in \pi_{l^1}^{in}$. Therefore, applying Property 7.8.6 on loop l^1, for any execution graph node j and for any iteration n of loop l^1, we get $O_j^{in}(m', n) \subseteq O_j^{in}(m'', n)$. Therefore, O_i^{exit} for any $i \in \pi_{l^1}^{out}$ (recall that O_i^{exit} represents the bus context exiting the loop l^1 from node i) computed at m''-th iteration of loop l is an over-approximation of O_i^{exit} computed at m'-th iteration of loop l.

Induction step Assume our claim holds for all basic blocks i having $n_i \leq k$. Therefore, $O_j^{in}(m') \subseteq O_j^{in}(m'')$ for any execution graph node j that corresponds to the instructions of any basic block i (having $n_i \leq k$). Moreover, for any inner loop l^k and for all $j \in \pi_{l^k}^{out}$, O_j^{exit} computed at m''-th iteration of l is always an over-approximation of O_j^{exit} computed at m'-th iteration of l,

We shall prove that our claim holds for all basic blocks having $n_i = k + 1$. As described in the preceding, we have the two following cases:

- (Case I) i is a basic block which is immediately enclosed by loop l.
- (Case II) i is a basic block which is immediately enclosed by some loop l^{k+1}, where l^{k+1} is the loop contained inside l and k different loops inside l precedes l^{k+1} in topological order.

For Case I, using our methodology for proving Properties 7.8.3-7.8.4, we can easily show that if the bus context for some basic block i (having $n_i = k+1$) at iteration m'' is not an *over-approximation* of the bus context of the same basic block at iteration m', it could be due to any of the three following reasons:

- The bus context at iteration m'' is not an *over-approximation* of the bus context at iteration m' for some basic block j having $n_j \leq k$, contradicting our induction hypothesis;
- There exists some loop l^x which appears prior to i in topological order but O_i^{exit} computed at m''-th iteration of loop l is not an over-approximation of O_i^{exit} computed at m'-th iteration of loop l for some $i \in \pi_{l^x}^{out}$. Since l^x appears prior in topological order of i, $x \leq k$. This also violates our induction hypothesis.
- For some $xi \in \pi_l^{in}$, $O_{xi}^{in}(m') \not\subseteq O_{xi}^{in}(m'')$, contradicting our assumption.

Now consider Case II. Assume $O_j^{in}(x, x')$ ($O_j^{out}(x, x')$) represents the incoming (outgoing) bus context at the execution graph node j at x-th iteration of l and at x'-th iteration of l^{k+1}. According to our induction hypothesis and the argument provided above, we get $O_j^{in}(m', 1) \subseteq O_j^{in}(m'', 1)$ for all $j \in \pi_{l^{k+1}}^{in}$. Therefore, applying Property 7.8.6 on loop l^{k+1}, for any execution graph node j and for any iteration n of loop l^{k+1}, we get $O_j^{in}(m', n) \subseteq O_j^{in}(m'', n)$. Consequently, for any $i \in \pi_{l^{k+1}}^{out}$, O_i^{exit} computed at m''-th iteration of loop l is an over-approximation of O_i^{exit} computed at m'-th iteration of loop l. This completes our induction.

Finally, we conclude that $O_j^{in}(m') \subseteq O_j^{in}(m'')$ for any execution graph node j that corresponds to some instruction in l. Consequently, $O_{xi}^{in}(m'+1) \subseteq O_{xi}^{in}(m''+1)$ for all $xi \in \pi_l^{in}$.

From the above argument, it is now straight-forward to see that the property also holds for any nested loop by proving the claims in a bottom up fashion of loop nests (*i.e.* an induction on the level of loop nests starting from the innermost loop). □

Property 7.8.8. (Termination Property) Consider two instructions p and q of basic block B. $(p, q) \in CR$ if and only if p and q may contend for the same functional unit. CR is called

the contention relation. Assume CR_n represents the contention relation at n-th iteration of pipeline modeling. Set of elements in CR_n monotonically decreases across different iterations n of pipeline modeling.

Proof. We prove the above claim by induction on number of iterations taken by the pipeline modeling. For some execution graph node i, assume $O_i^{in}(n)$ ($O_i^{out}(n)$) represents the incoming (outgoing) bus context at iteration n. Also assume $earliest[t_i^{ready}, n]$ ($latest[t_i^{finish}, n]$) represents the earliest (latest) ready (finish) time of execution graph node i at iteration n.

Base case $n = 1$. We start with all possible pairs of instructions in the contention relation (*i.e.* we assume that every pair of instructions which may use same functional unit can potentially delay each other). Therefore, the set of elements in the contention relation *trivially* decreases after the first iteration (*i.e.* $CR_2 \subseteq CR_1$).

Induction step We assume that $CR_n \subseteq CR_{n-1}$ and we shall prove that $CR_{n+1} \subseteq CR_n$. We prove the same by contradiction (*i.e.* assume that $CR_{n+1} \nsubseteq CR_n$). Informally, we have at least two execution graph nodes i and j which have *disjoint* timing intervals at iteration n but *overlapping* timing intervals at iteration $n + 1$. This is only possible if one of the following conditions hold:

- $earliest[t_i^{ready}, n{+}1] < earliest[t_i^{ready}, n]$ (or $earliest[t_j^{ready}, n{+}1] < earliest[t_j^{ready}, n]$).
- $latest[t_i^{finish}, n + 1] > latest[t_i^{finish}, n]$ (or $latest[t_j^{finish}, n + 1] > latest[t_j^{finish}, n]$).

However, above situation may arise only if one of the following two conditions hold: 1) $O_k^{in}(n{+}1) \nsubseteq O_k^{in}(n)$ for some execution graph node k. Since $CR_n \subseteq CR_{n-1}$, Property 7.8.4 ensures $O_p^{in}(n + 1) \nsubseteq O_p^{in}(n)$ for at least one node p which does not have any predecessor. This is a contradiction as $O_p^{in}(n + 1) = O_p^{in}(n)$. 2) $CR_n \nsubseteq CR_{n-1}$, which leads to more contention at n-th iteration and thereby increasing $latest[t_i^{finish}, n + 1]$ for some node i. However, this is a contradiction of our induction hypothesis.

This property ensures that our iterative framework always terminates in the presence of shared cache and shared bus. □

Property 7.8.9. Computation of O_i^{in} and O_i^{out} is always *sound*.

Proof. This follows directly from the previous properties. Property, 7.8.6 ensures that we include all possible contexts for a basic block inside loop. Equation 7.10 and Equation 7.11 ensure that we include all possible TDMA offsets from different program paths. As contention decreases monotonically over different iterations of pipeline modeling (Property 7.8.8), Equation 7.9 and Equation 7.7 ensure that the value of O_i^{in} and O_i^{out} are *sound* over-approximations of respective bus contexts. Finally, the soundness of the analysis presented in [41] guarantees that we always compute an overapproximation of bus contexts at loop exit.

Essentially, we show that the search space of possible bus contexts is never pruned throughout the program. Therefore, our analysis maintain *soundness* when a *lower bus delay* may lead to global worst case scenario. □

Finally, we conclude that the longest acyclic path search in the execution graph always results in a *sound* estimation of basic block WCET. Moreover, we are able to consider an overapproximation of all possible bus contexts if a basic block executes with multiple bus contexts (Properties 7.8.6 -7.8.7). The IPET approach, on the other hand, searches for the *longest feasible program path* to ensure a *sound* estimation of whole program's WCET.

7.9 Experimental evaluation

Experimental setup

We have chosen moderate to large size benchmarks from [2], which are generally used for timing analysis. The code size of the benchmarks ranges from 2779 bytes (`bsort100`) to 118351 bytes (`nsichneu`), with an average code size of 18500 bytes. Individual benchmarks are compiled into simplescalar PISA (Portable Instruction Set Architecture) [81] — a MIPS like instruction set architecture. We use the simplescalar gcc cross compiler with optimization level `-O2` to generate the PISA compliant binary of each benchmark. The control flow graph (CFG) of each benchmark is extracted from its PISA compliant binary and is used as an input to our analysis framework.

To validate our analysis framework, the simplescalar toolset [81] was extended to support the simulation of shared cache and shared bus. The simulation infrastructure is used to compare the estimated WCET with the observed WCET. Observed WCET is measured by simulating the program for a few program inputs. Nevertheless, we would like to point out that the presence of a shared cache and a shared bus makes the realization of the worst case scenario extremely

Table 7.1: Default micro-architectural setting for experiments

Component	*Default settings*	*Perfect settings*
Number of cores	2	NA
pipeline	1-way, inorder 4-entry IFQ, 8-entry ROB	NA
L1 instruction cache	2-way associative, 1 KB miss penalty = 6 cycles	All accesses are L1 *hit*
L2 instruction cache	4-way associative, 4 KB miss penalty = 30 cycles	NA
Shared bus	slot length = 50 cycles	Zero bus delay
Branch predictor	2 level predictor, L1 size=1 L2 size=4, history size=2	Branch prediction is *always correct*

challenging. In the presence of a shared cache and a shared bus, the worst case scenario depends on the interleavings of threads, which are running on different cores. Consequently, the observed WCET result in our experiments may sometimes highly under-approximate the *actual WCET*.

For all of our experiments, we present the WCET overestimation ratio, which is measured as $\frac{Estimated\ WCET}{Observed\ WCET}$. For each reported overestimation ratio, the system configuration during the analysis (which computes $Estimated\ WCET$) and the measurement (which computes $Observed\ WCET$) are kept *identical*. Unless otherwise stated, our analysis uses the default system configuration in Table 7.1 (as shown by the column "Default settings"). Since the data cache modeling is not yet included in our current implementation, all data accesses are assumed to be *L1 cache hits* (for analysis and measurement both).

To check the dependency of WCET overestimation on the type of conflicting task (being run in parallel on a different core), we use two different tasks to generate the inter-core conflicts — 1) `jfdctint`, which is a single path program and 2) `statemate`, which has a huge number of paths. In our experiments (Figures 7.5-7.7), we use `jfdctint` to generate inter-core conflicts to the first half of the tasks (*i.e.* `matmult` to `nsichneu`). On the other hand, we use `statemate` to generate inter-core conflicts to the second half of the tasks (*i.e.* `edn` to `st`). Due to the absence of any infeasible program path, inter-core conflicts generated by a single path program (*e.g.* `jfdctint`) can be more accurately modeled compared to a multi-path program (*e.g.* `statemate`). Therefore, in the presence of a shared cache, we expect a better WCET overestimation ratio for the first half of the benchmarks (*i.e.* `matmult` to `nsichneu`) compared to the second half (*i.e.* `edn` to `st`).

To measure the WCET overestimation due to *cache sharing*, we compare the WCET result

with two different design choices, where the level 2 cache is partitioned. For a two-core system, two different partitioning choices are explored: first, each partition has the same number of cache sets but has half the number of ways compared to the original shared cache (called *vertical* partitioning). Secondly, each partition has half the number of cache sets but has the same number of ways compared to the original shared cache (called *horizontal* partitioning). In our default configuration, therefore, each core is assigned a 2-way associative, 2 KB L2 cache in the *vertical partitioning*, whereas each core is assigned a 4-way associative, 2 KB L2 cache in the *horizontal partitioning*.

Finally, to pinpoint the source of WCET overestimation, we can selectively turn off the analysis of different micro-architectural components. We say that a micro-architectural component has *perfect setting* if the analysis of the same is turned off (refer to column "Perfect settings" in Table 7.1).

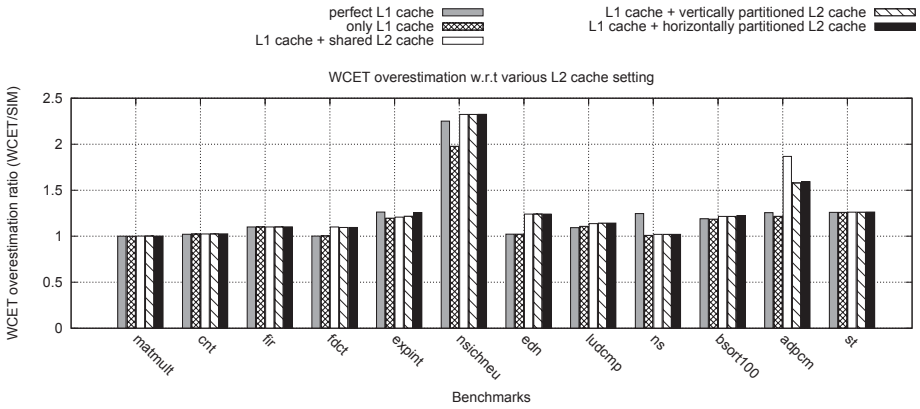


Figure 7.5: Effect of shared and partitioned L2 cache on WCET overestimation

Basic analysis result

Effect of caches Figure 7.5 shows the WCET overestimation ratio with respect to different L1 and L2 cache settings in the presence of a *perfect* branch predictor and a *perfect* shared bus. Results show that we can reasonably bound the WCET overestimation ratio except for `nsichneu`. The main source of WCET overestimation in `nsichneu` comes from the *path analysis* and *not* due to the micro-architectural modeling. This is expected, as `nsichneu` contains more than two hundred branch instructions and many *infeasible paths*. These infeasible paths can be eliminated by providing additional user constraints into our framework and hence improving the result. We also observe that the partitioned L2 caches may lead to a better WCET

overestimation compared to the shared L2 caches, with the *vertical* L2 cache partitioning almost always working as the best choice. The positive effect of the vertical cache partitioning is visible in `adpcm`, where the overestimation in the presence of a shared cache rises. This is due to the difficulty in modeling the inter-core cache conflicts from `statemate` (a *many-path* program being run in parallel).

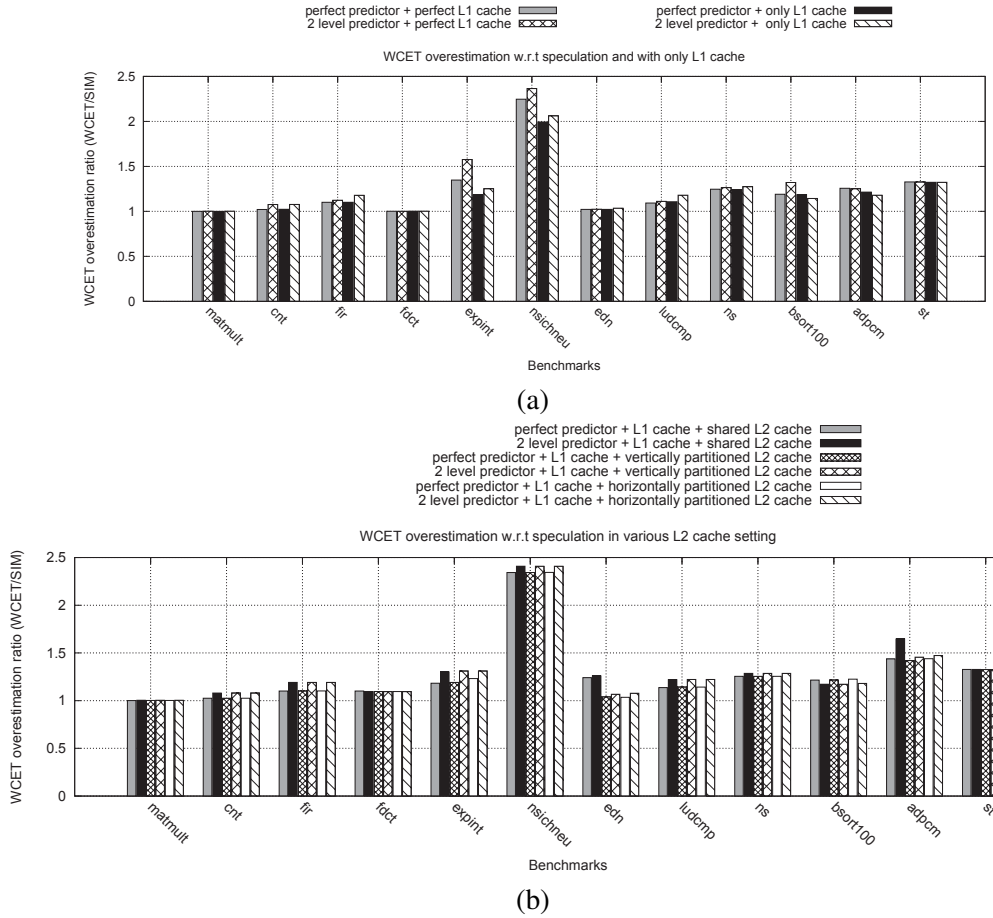


Figure 7.6: (a) Effect of speculation on L1 cache, (b) effect of speculation on partitioned and shared L2 caches

Effect of speculative execution As we explained in Section 7.6, the presence of a branch predictor and speculative execution may introduce additional computation cycles for executing a mispredicted path. Moreover, speculative execution may introduce additional cache conflicts from a mispredicted path. The results in Figure 7.6(a) and Figure 7.6(b) show the effect of speculation in L1 and L2 cache, respectively. Mostly, we do not observe any sudden *spikes* in the WCET overestimation just due to speculation. `adpcm` shows some reasonable increase in WCET overestimation with L2 caches and in the presence of speculation (Figure 7.6(b)).

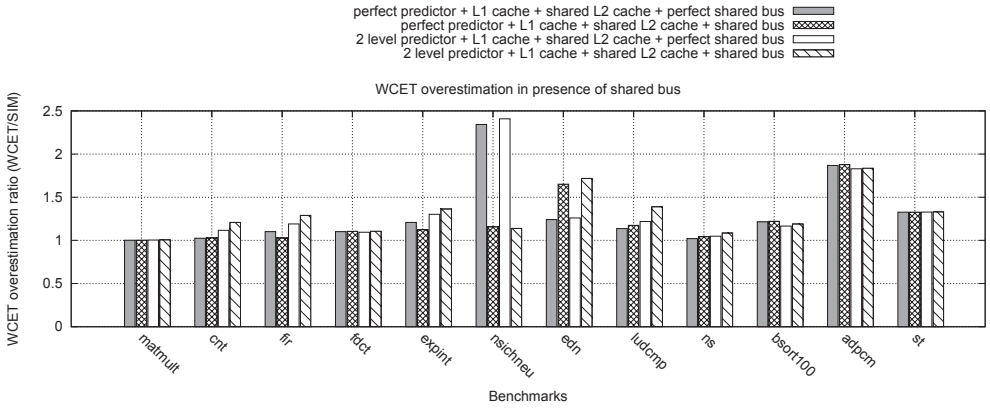


Figure 7.7: Effect of shared bus on WCET overestimation

This increase in the overestimation ratio can be explained from the overestimation arising in the modeling of the effect of speculation in cache (refer to Section 7.6). Due to the abstract *join* operation to combine the cache states in correct and mispredicted path, we may introduce some *spurious* cache conflicts. Nevertheless, our approach for modeling the speculation effect in cache is scalable and produces tight WCET estimates for most of the benchmarks.

Effect of shared bus Figure 7.7 shows the WCET overestimation in the presence of a shared cache and a shared bus. We observe that our shared bus analysis can reasonably control the overestimation due to the shared bus. Except for `edn` and `nsichneu`, the overestimation in the presence of a shared cache and a shared bus is mostly equal to the overestimation when shared bus analysis is turned off (*i.e.* a *perfect* shared bus). Recall that each overestimation ratio is computed by performing the analysis and the measurement on *identical system configuration*. Therefore, the analysis and the measurement both includes the shared bus delay only when the shared bus is *enabled*. For a *perfect shared bus setting*, both the analysis and the measurement consider a *zero latency* for all the bus accesses. As a result, we also observe that our shared bus analysis might be more accurate than the analysis of other micro-architectural components (*e.g.* in case of `nsichneu`, `expint` and `fir`, where the WCET overestimation ratio in the presence of a shared bus might be less than the same with a *perfect* shared bus). In particular, `nsichneu` shows a drastic fall in the WCET overestimation ratio when the shared bus analysis is *enabled*. For `nsichneu`, we found that the execution time is dominated by shared bus delay, which is most accurately computed by our analysis for this benchmark. On the other hand, we observed in Figure 7.5 that the main source of WCET overestimation in `nsichneu` is *path analysis*, due to the presence of many infeasible paths. Consequently, when shared bus analysis

is turned off, the overestimation arising from path analysis dominates and we obtain a high WCET overestimation ratio. Average WCET overestimation in the presence of both a shared cache and a shared bus is around 50%.

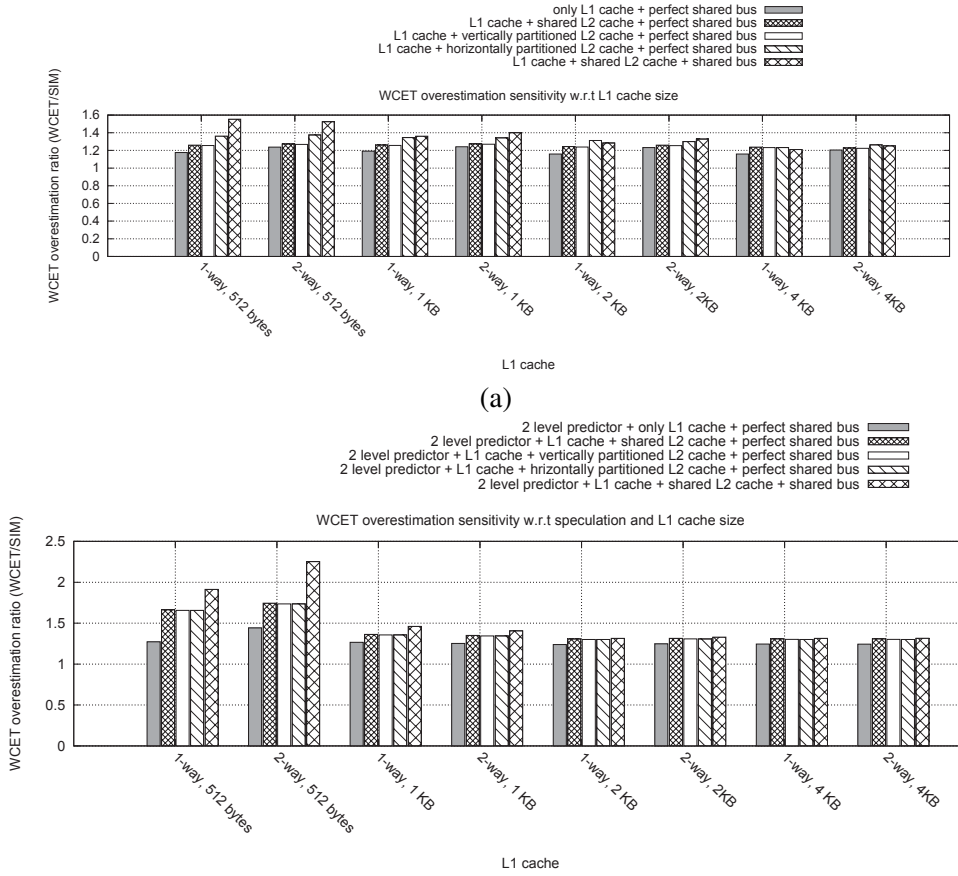


Figure 7.8: WCET overestimation sensitivity w.r.t. L1 cache (a) without speculation, (b) with speculation

WCET analysis sensitivity w.r.t. micro-architectural parameters

In this section, we evaluate the WCET overestimation sensitivity with respect to different micro-architectural parameters. For the following experiments, the reported WCET overestimation denotes the *geometric mean* of the term $\frac{Estimated\ WCET}{Observed\ WCET}$ over all the different benchmarks.

WCET sensitivity w.r.t. L1 cache size Figure 7.8(a) and Figure 7.8(b) show the *geometric mean* of WCET overestimation for different L1 cache sizes, with and without speculation, respectively. To keep the L2 cache bigger than the L1 cache, total L2 cache is kept at 4-way, 16 KB for all the experiments in Figures 7.8(a)-(b). Therefore, for *horizontally* and *vertically*

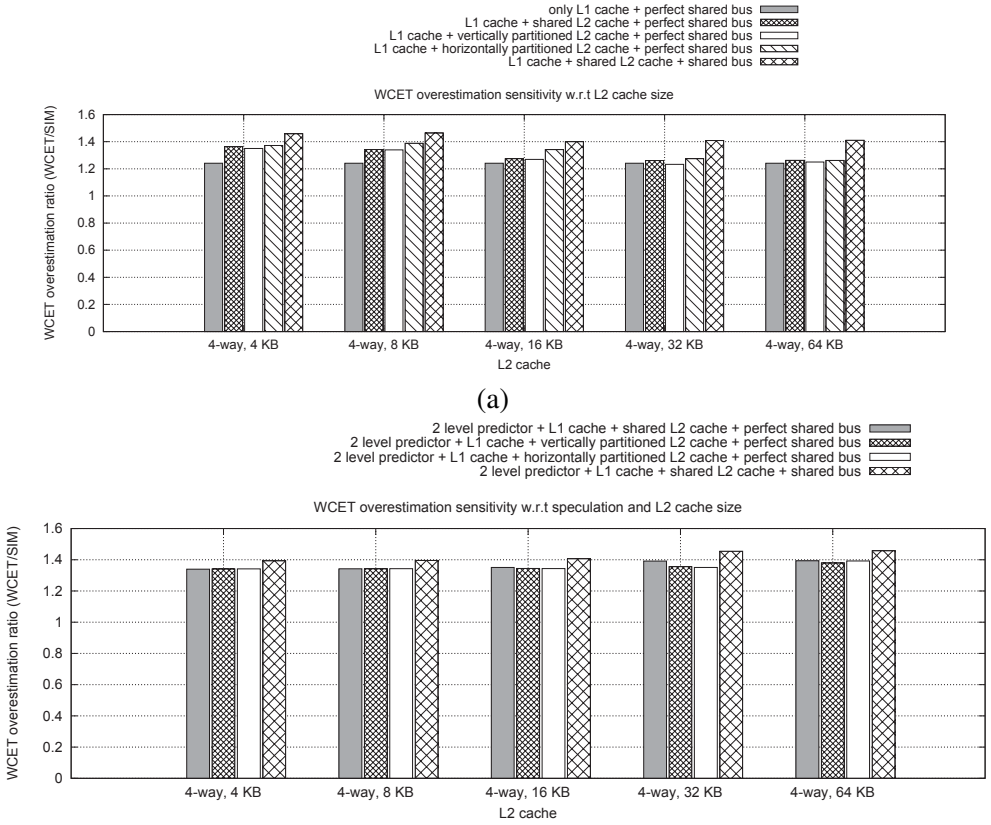
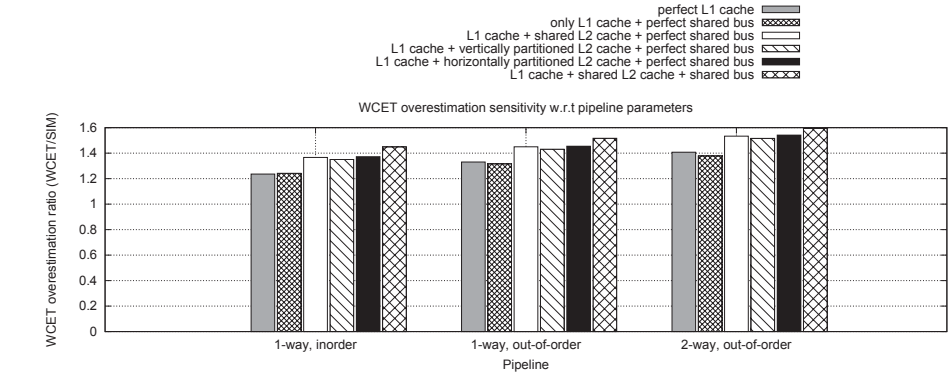


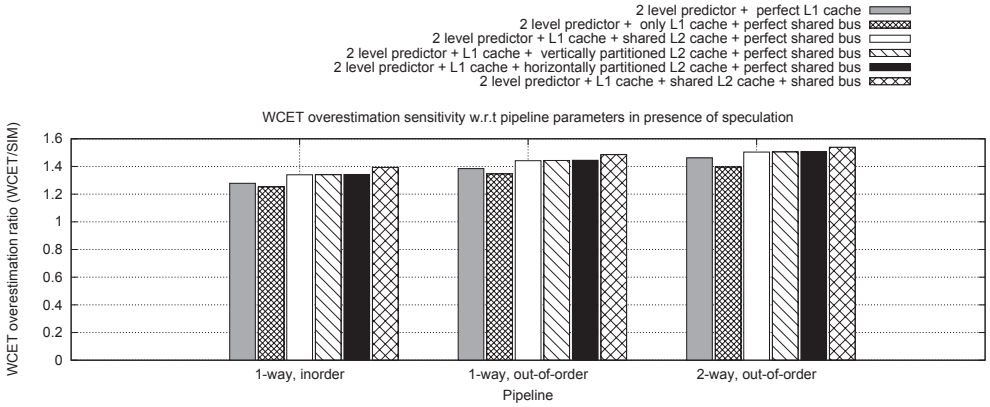
Figure 7.9: WCET overestimation sensitivity w.r.t. L2 cache (a) without speculation, (b) with speculation

partitioned L2 cache architectures, each core uses an 8 KB L2 cache. Naturally, in the presence of speculation, the overestimation is slightly higher. However, our framework is able to maintain an average overestimation ratio around 20% without speculation and around 40% with speculation.

WCET sensitivity w.r.t. L2 cache size Figure 7.9(a) and Figure 7.9(b) show the *geometric mean* of WCET overestimation for different L2 cache sizes, with and without speculation, respectively. On average, WCET overestimation in the presence of shared L2 cache is higher compared to partitioned L2 cache architectures. As pointed out earlier, this is due to the inherent difficulties in modeling the inter-core cache conflicts. Nevertheless, our analysis framework captures an average overestimation around 40% (50%) without (with) speculation over different L2 cache settings.



(a)



(b)

Figure 7.10: WCET overestimation sensitivity w.r.t. different pipelines (a) without speculation, (b) with speculation

WCET sensitivity w.r.t. different pipelines We have done experiments for different pipelines. Figure 7.10(a) (without speculation) and Figure 7.10(b) (with speculation) show the WCET overestimation sensitivity for inorder, out-of-order and superscalar pipelines. Superscalar pipelines increase the instruction level parallelism and so as the performance of entire program. However, it also becomes difficult to model the inherent instruction level parallelism in the presence of superscalar pipelines. Therefore, Figure 7.10(a) and 7.10(b) both show an increase in WCET overestimation with superscalar pipelines. However, it is clear from both the figures that the additional overestimation mostly comes from the superscalar pipeline modeling (results marked by "without cache" and "2lev without cache" respectively) and *not* from the modeling of caches.

WCET sensitivity w.r.t. bus slot length Finally, we show how the WCET overestimation is affected with respect to bus slot length. Figure 7.11 shows the WCET overestimation sensitivity with respect to different bus slot lengths. With very high bus slot lengths (*e.g.* 70 or 80 cycles), WCET overestimation normally increases (as shown in Figure 7.11). This is due to the fact that

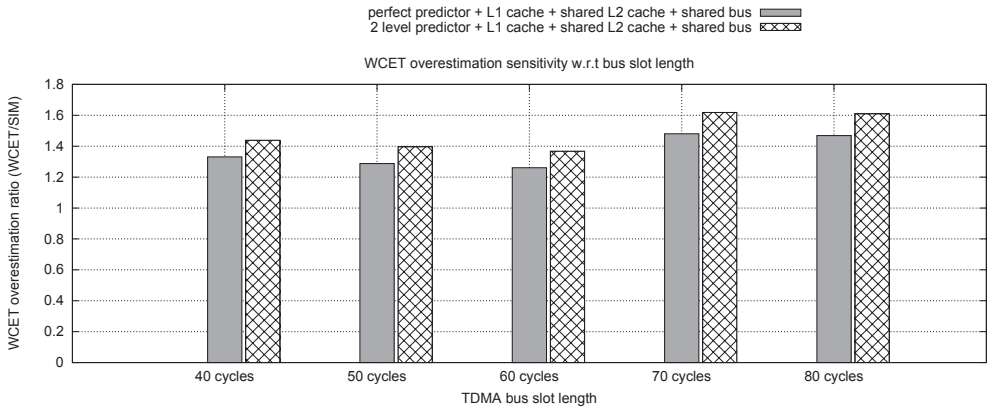


Figure 7.11: WCET overestimation sensitivity w.r.t. different bus slot length (with and without speculative execution)

with higher bus slot lengths, the search space for possible bus contexts (or set of TDMA offsets) increases. As a result, it is less probable to expose the worst case scenario in simulation with higher bus slot lengths.

Analysis time

We have performed all the experiments on an 8 core, 2.83 GHz Intel Xeon machine having 4 GB of RAM and running Fedora Core 4 operating system. Table 7.2 reports the *maximum* analysis time when the shared bus analysis is *disabled* and Table 7.3 reports the *maximum* analysis time when all the analyses are enabled (*i.e.* cache, shared bus and pipeline). Recall from Section 8.2 that our WCET analysis framework is broadly composed of two different parts, namely, micro-architectural modeling and implicit path enumeration (IPET) through integer linear programming (ILP). The column labeled "μ arch" captures the time required for micro-architectural modeling. On the other hand, the column labeled "ILP" captures the time required for path analysis through IPET.

In the presence of speculative execution, number of mispredicted branches is modeled by integer linear programming [18]. Such an ILP-based branch predictor modeling, therefore, increases the number of constraints which need to be considered by the ILP solver. As a result, the ILP solving time increases in the presence of speculative execution (as evidenced by the second rows of both Table 7.2 and Table 7.3).

Shared bus analysis increases the micro-architectural modeling time (as evidenced by Table 7.3) and the analysis time usually increases with the bus slot length. The time for the shared bus analysis generally appears from tracking the bus context at different pipeline stages. A higher

bus slot length usually leads to a higher number of bus contexts to analyze, thereby increasing the analysis time.

In Table 7.2 and Table 7.3, we have only presented the analysis time for the longest running benchmark (`nsichneu`) from our test-suite. For any other program used in our experiments, the entire analysis (micro-architectural modeling and ILP solving time) takes around 20-30 seconds on average to finish.

The results reported in Table 7.2 show that the ILP-based modeling of branch predictor usually increases the analysis time. Therefore, for a more efficient but less precise analysis of branch predictors, one can explore different techniques to model branch predictors, such as abstract interpretation. Shared bus analysis time can be reduced by using different offset abstractions, such as *interval* instead of an *offset set*. Nevertheless, the appropriate choice of analysis method and abstraction depends on the precision-scalability tradeoff required by the user.

Table 7.2: Analysis time [of `nsichneu`] in *seconds*. The first row represents the analysis time when speculative execution was *disabled*. The second row represents the time when speculative execution was *enabled*

Shared L2 cache										Pipeline					
4 KB		8 KB		16 KB		32 KB		64 KB		1-way inorder		1-way out-of-order		2-way superscalar	
μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP
1.2	1.3	1.4	1.3	1.7	1.3	2.3	1.3	4.8	1.2	1.3	1.3	1.2	1.3	1.3	1.4
2.6	240	2.9	240	3.5	238	4.6	238	7	239	2.6	238	2.4	239	2.8	254

Table 7.3: Analysis time [of `nsichneu`] in *seconds*. The first row represents the analysis time when speculative execution was *disabled*. The second row represents the time when speculative execution was *enabled*

TDMA bus slot length									
40 cycles		50 cycles		60 cycles		70 cycles		80 cycles	
μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP
75.8	4	100	4	128	4	160	4.2	198	5.1
128	162	163	156	205	158	261	181	363	148

7.10 Extension of shared cache analysis

Our discussion on cache analysis has so far concentrated on the *least-recently-used* (LRU) cache replacement policies. However, a widely used cache replacement policy is *first-in-first-out*

(FIFO). FIFO cache replacement policy has been used in embedded processors such as ARM9 and ARM11 [84]. Recently, abstract interpretation based analysis of FIFO replacement policy has been proposed in [85; 86] for single level caches and for multi-level caches in [87]. In this section, we shall discuss the extension of our shared cache analysis for FIFO cache replacement policy. We shall also show that such an extension will not change the modeling of timing interactions among shared cache and other basic micro-architectural components (*e.g.* pipeline and branch predictor).

7.10.1 Review of cache analysis for FIFO replacement

We use the *must cache analysis* for FIFO replacement as proposed in [85]. In FIFO replacement, when a cache set is full and still the processor requests fresh memory blocks (which map to the same cache set), the *first* cache line entering the respective cache set (*i.e.* first-in) is replaced. Therefore, the set of *tags* in a k-way FIFO abstract cache set (say $\mathcal{A}_s$) can be arranged from last-in to first-out order ([85]) as follows:

$$\mathcal{A}_s = [T_1, T_2, \ldots, T_k] \tag{7.21}$$

where each $T_i \subseteq \mathcal{T}$ and $\mathcal{T}$ is the set of all cache tags. Unlike LRU, cache state never changes upon a *cache hit* with FIFO replacement policy. Therefore, the cache state update on a memory reference depends on the *hit-miss* categorization of the same memory reference. Assume that a memory reference belongs to cache tag tag_i. The FIFO abstract cache set $\mathcal{A}_s = [T_1, T_2, \ldots, T_k]$ is updated on the access of tag_i as follows:

$$\tau([T_1, T_2, \ldots, T_k], tag_i) = \begin{cases} [T_1, T_2, \ldots, T_k], & \text{if } tag_i \in \bigcup_i T_i; \\ [\{tag_i\}, T_2, \ldots, T_{k-1}], & \text{if } tag_i \notin \bigcup_i T_i \wedge |\bigcup_i T_i| = k; \\ [\phi, T_2, \ldots, T_{k-1} \cup \{tag_i\}], & \text{otherwise.} \end{cases} \tag{7.22}$$

The first scenario captures a *cache hit* and the second scenario captures a *cache miss*. Third scenario appears when the static analysis cannot accurately determine the *hit-miss* categorization of the memory reference.

The *abstract join* function for the FIFO must cache analysis is exactly same as the LRU must cache analysis. The join function between two abstract FIFO cache sets computes the

intersection of the abstract cache sets. If a cache tag is available in both the abstract cache sets, the *right most* relative position of the cache tag is captured after the join operation.

7.10.2 Analysis of shared cache with FIFO replacement

We implement the *must cache analysis* for FIFO replacement as described in the preceding. To distinguish the cold cache misses at the first iterations of loops and different procedure calling contexts, our cache analysis employs the *virtual-inline-virtual-unrolling* (VIVU) approach (as described in [9]). After analyzing the L1 cache memory references are categorized as *all-hit* (AH), *all-miss* (AM) or *unclassified* (NC). AM and NC categorized memory references may access the L2 cache and therefore, the L2 cache state is updated for the memory references which are categorized AM or NC in the L1 cache (as in [87]).

To analyze the shared cache, we used our previous work on shared cache [15] for LRU cache replacement policy. [15] employs a separate shared cache conflict analysis phase. For FIFO replacement policy too, we can use the exactly same idea to analyze the set of inter-core cache conflicts. Shared cache conflict analysis may change the categorization of a memory reference from all-hit (AH) to unclassified (NC). For the sake of illustration, assume a memory reference which accesses the memory block m. This analysis phase first computes the number of unique conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially replace m from shared cache. More precisely, for an N-way set associative cache, hit/miss categorization (CHMC) of corresponding memory reference is changed from all-hit (AH) to unclassified (NC) if and only if the following condition holds:

$$N - AGE_{fifo}(m) < |\mathcal{M}_c(m)| \tag{7.23}$$

where $|\mathcal{M}_c(m)|$ represents the number of conflicting memory blocks from different cores which may potentially access the same L2 cache set as m. $AGE_{fifo}(m)$ represents the relative position of memory block m in the FIFO abstract cache set and in the absence of inter-core cache conflicts. Recall that the memory blocks (or the tags) are arranged according to the *last-in* to *first-out* order in the FIFO abstract cache set. Therefore, the term $N - AGE_{fifo}(m)$ captures the maximum number of fresh memory blocks which can enter the FIFO cache before m being evicted out.

7.10.3 Interaction of FIFO cache with pipeline and branch predictor

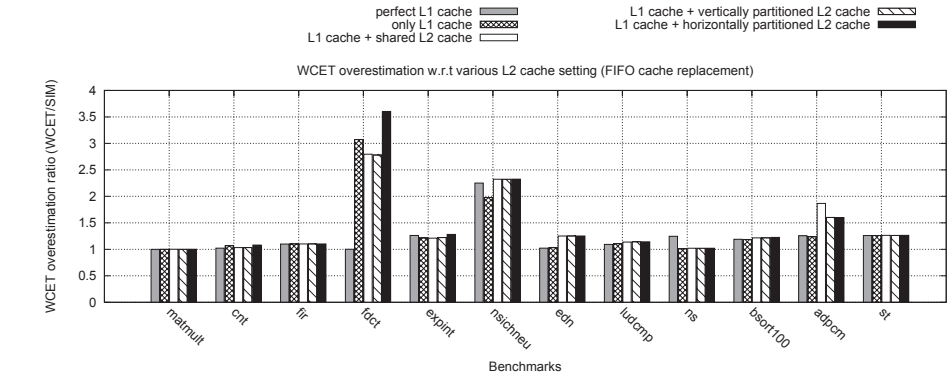
As described in the preceding, after the FIFO shared cache analysis, memory references are categorized as *all-hit* (AH), *all-miss* (AM) or *unclassified* (NC). In the presence of pipeline, such a categorization of instruction memory references add computation cycle with the instruction fetch (IF) stage. Therefore, we use Equation 7.1 to compute the latency suffered by cache hit/miss and propagate the latency through different pipeline stages.

Recall from Section 7.6.1 that speculative execution may introduce additional cache conflicts. In Section 7.6.1, we proposed to modify the abstract interpretation based cache analysis to handle the effect of speculative execution on cache. From Figure 7.4, we observe that our solution is *independent* of the cache replacement policies concerned. Our proposed modification performs an *abstract join* operation on the cache states along the *correct* and *mispredicted* path (as shown in Figure 7.4). Therefore, for FIFO replacement polices the abstract join operation is performed according to the FIFO replacement analysis (instead of LRU join operation we performed in case of LRU caches).

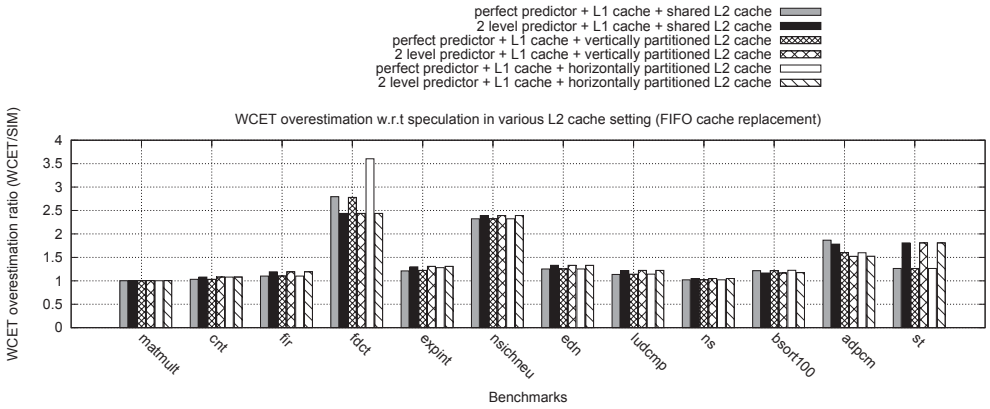
7.10.4 Experimental result

Figure 7.12 demonstrates our WCET analysis experience with FIFO replacement policy. We have used the exactly same experimental setup as mentioned in Section 7.9. Figure 7.12(a) shows the WCET overestimation ratio in the absence of speculative execution and Figure 7.12(a) shows the same in the presence of branch predictor. In general, our analysis framework can reasonably bound the WCET overestimation for FIFO cache replacement, except for `fdct`. Such an overestimation for `fdct` is solely due to the presence of a FIFO cache and not due to the presence of cache sharing, as clearly evidenced by Figure 7.12(a). However, as mentioned in [88], the observed worst-case for FIFO replacement may highly under-approximate the *true worst case* due to the *domino effect*. Otherwise, our results in Figure 7.12(a) show that FIFO is a reasonably good alternative of LRU replacement even in the context of shared caches.

Figure 7.12(b) shows that our modeling of the interaction between FIFO cache and the branch predictor does not much affect the WCET overestimation. As evidenced by Figure 7.12(b), the increase in the WCET overestimation is *minimal* due to the speculation.



(a)



(b)

Figure 7.12: Analysis of cache in the presence of FIFO replacement policy (a) WCET overestimation w.r.t. different L2 cache architectures, (b) WCET overestimation in the presence of FIFO cache and speculative execution

7.10.5 Other cache organizations

In the preceding, we have discussed the extension of our WCET analysis framework with FIFO replacement policy. We have shown that as long as the cache tags in an abstract cache set can be arranged according to the order of their replacement, our shared cache conflict analysis can be integrated. As a result, our modeling for the timing interaction among (shared) cache, pipeline and branch predictor is independent of the underlying cache replacement policy. Nevertheless, for some cache replacement policies, arranging the cache tags according to the order of their replacement poses a challenge (*e.g.* PLRU [89]). Cache analysis based on *relative competitiveness* [84] tries to analyze a cache replacement policy with respect to an equivalent LRU cache, but with different parameters (*e.g.* associativity). Any cache replacement analysis based on relative competitiveness can directly be integrated with our WCET analysis framework. Nevertheless, more precise analysis than the ones based on relative competitiveness can be designed, as shown in [89] for PLRU policy. However, we believe that designing more precise cache analysis is out-

side the scope of this paper. The purpose of our work is to propose a unified WCET analysis framework and any precision gain in the existing cache analysis technique will directly benefit our framework by improving the precision of WCET prediction.

In this paper, we have focused on the *non-inclusive* cache hierarchy. In multi-core architectures, inclusive cache hierarchy may limit performance when the size of the largest cache is not significantly larger than the sum of the smaller caches. Therefore, processor architects sometimes resort to non-inclusive cache hierarchies [90]. On the other hand, inclusive cache hierarchies greatly simplify the cache coherence protocol. The analysis of inclusive cache hierarchy requires to take account of the *invalidations* of certain cache lines to maintain the *inclusion* property (as shown in [87] for multi-level private cache hierarchies). The analysis in [87] first analyzes the multi-level caches for general *non-inclusive* cache hierarchies and a post-processing phase may change the categorization of a memory reference from *all-hit* (AH) to *unclassified* (NC). Our shared cache conflict analysis phase can be applied on this reduced set of AH categorized memory reference for inclusive caches, keeping the rest of our WCET analysis framework entirely unchanged. Therefore, we believe that the inclusive cache hierarchies do not pose any additional challenge in the context of shared caches and the analysis of such cache hierarchies can easily be integrated, keeping the rest of our WCET analysis framework unchanged.

7.11 Chapter summary

In this chapter, we have proposed a *sound* WCET analysis framework by modeling different micro-architectural components and their interactions in a multi-core processor. Our analysis framework is also *sound* in the presence of *timing anomalies*. We have performed a detailed evaluation of our proposed WCET analysis framework. Our experiments suggest that we can obtain tight WCET estimates for the majority of benchmarks in a variety of micro-architectural configurations. Apart from design space exploration, we believe that our framework can be used to figure out the major sources of overestimation in multi-core WCET analysis. As a result, our framework can help in designing predictable hardware for real-time applications and it can also help writing real-time applications for the predictable execution in multi-cores. More details about the multi-core analyzer and the simulator are available in [91].

Chapter 8

Cache Related Preemption Delay Analysis for Shared Cache

In previous chapters, we have assumed an uninterrupted execution of each task in each core. However, real-time systems are often multi-tasking and the execution of a task can be interrupted (or preempted) by a different task. In this Chapter, we shall extend our multi-core WCET analysis framework for a multi-tasking application setup. Specifically, we shall look at the problem of statically predicting the additional cache miss penalty in the presence of shared caches.

8.1 Introduction

Caches have a key role to play for enhancing performance of any running application on the underlying hardware platform. On the other hand, employing caches introduces additional complications to analyze the effect of *intra-task* and *inter-task* interferences on cache. Literature on static cache analysis handle the problem of *intra-task* interferences on cache. *Inter task* interferences on cache are created due to preemption. Suppose a low priority task t is preempted by a higher priority task t', the set of cache blocks used by t' is $\mathcal{C}_{t'}$ and the set of cache blocks used by t before the preemption took place is denoted by $\mathcal{C}_t$. If $\mathcal{C}_t \cap \mathcal{C}_{t'} \neq \phi$, t' may replace some of the cache blocks used by t and therefore may introduce additional cache misses to t when it resumes. This variety of inter-task interference on cache performance is well known in literature as *cache related preemption delay* (CRPD).

Research for statically predicting CRPD has been done in the past few years [28; 29; 30; 31;

33; 34]. All prior works on CRPD consider a single level instruction or data cache. However, with the advent of complex hardware in real time embedded systems (e.g., cache hierarchies, multi-core), many processors (*e.g.*, ARM) employ a bigger level two (L2) cache for improving the performance. Moreover, in multi-core architectures, the last level of cache hierarchy (typically the second level) is shared among all the cores (e.g., ARM MPCORE). Therefore, there is a need to consider cache hierarchies for estimating the inter-task interferences. In this chapter, we propose a CRPD analysis framework which can be applied to a two-level, non-inclusive cache hierarchy. More importantly, we propose an analysis framework which can be used in the presence of a shared cache, thereby providing a solution for computing CRPD in the current generation of multi-core architectures.

The key to estimate CRPD is based on the notion of *useful cache blocks* (UCB). UCB denotes a cache block which might be used by the preempted task after preemption. Therefore, the number of UCBs poses an upper bound on CRPD. Estimation can further be tightened by analyzing the *evicting cache blocks* (ECB) in the preempting task. ECB denotes a cache block which might be used by the preempting task. In the presence of non-inclusive cache hierarchy, some memory blocks in the preempted task may access the L2 cache *only* after the preemption — thereby increasing the amount of *intra-task cache interference* on the L2 cache. Therefore, in the presence of two level cache hierarchy, CRPD computation might be affected due to the variation in the *intra-task* L2 cache interference after preemption.

We show that in the presence of cache hierarchy, *a memory reference* may suffer multiple *L2 cache misses* after the preemption. Our framework gives reasonable theoretical bounds on the number of L2 cache misses suffered by the same memory reference after preemption. This theoretical bound, on the other hand, depends on the organization of cache hierarchy (in terms of the *number of cache sets* and the *associativity*). We propose a CRPD analysis framework which uses this bound and estimates the CRPD. Finally, we extend our CRPD analysis framework for shared caches in multi-cores by handling both the inter-core and inter-task cache conflicts.

We can guarantee the *correctness* of our CRPD analysis framework via formal proofs. We have also implemented our CRPD analysis framework into Chronos [23] - a freely available, open-source, WCET analysis tool. To experimentally validate our analysis framework, we also extend the simplescalar toolset [81] and observe the cache related preemption delay. We have evaluated our framework using a number of benchmarks from [2] and using different tasks from an *unmanned aerial vehicle* (UAV) control application. Experiments show that our framework

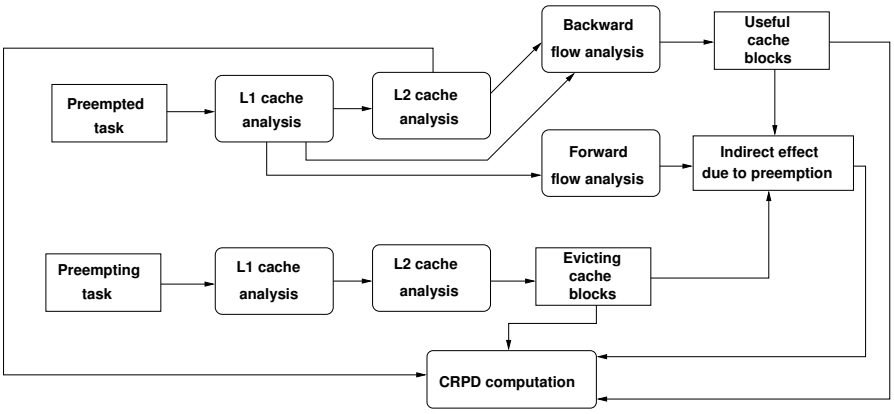


Figure 8.1: CRPD analysis framework

gives *precise* estimations for most of the benchmarks.

8.2 Overview of our analysis

In this section, we shall first give an overview of our CRPD analysis framework for a two level cache hierarchy. Subsequently, we shall discuss the *key challenges* in analyzing CRPD in the presence of level two caches. We shall show through a few examples that major changes are required in the existing CRPD analysis framework based on the concept of UCB and ECB. We shall also show through an example that a *sound* CRPD estimation is not possible solely using the concepts of UCB and ECB.

System model

In this work, we only model the effect of instruction memory. We assume a two-level instruction cache hierarchy (L1 and L2 cache). For multi-cores, we assume that each core has a private L1 cache and multiple cores can share an L2 cache. A memory block is always accessed from the L1 cache. If a memory reference *misses* in both the L1 and L2 cache, it is loaded from main memory to both the cache levels. On the other hand, if a memory block *misses* in the L1 cache but *hits* in the L2 cache, it is loaded into the L1 cache. Finally, the L2 cache is not accessed when a memory reference *hits* in the L1 cache. We assume a *LRU cache replacement policy* and we consider only *non-inclusive* caches.

Overall framework

Our CRPD analysis framework is shown in Figure 10.2. We first perform L1 and L2 cache analysis on the preempted task using [9] and [12], respectively. The outcome of L1 and L2 cache analysis is used by a *backward flow analysis*, which in turn derives the set of useful cache blocks (UCB) in the context of a two-level cache hierarchy. A similar L1 and L2 cache analysis on the preempting task derives the set of evicting cache blocks (ECB) in L1 and L2 cache. A separate *forward flow analysis* is used to estimate the additional *intra-task L2 cache conflicts* generated due to preemption. We call this additional intra-task L2 cache conflict as the *indirect effect of preemption* (as shown by the box labeled "indirect effect due to preemption" in Figure 10.2). Finally, the information derived by the backward and forward flow analysis are processed to compute the *cache related preemption delay* (CRPD) of the underlying preempted task. Our CRPD analysis does not account the cache misses already accounted by intra-task L1 and L2 cache analysis (similar to [33]). Therefore, the CRPD computed by our framework is *safe* only when considered together with the WCET analysis.

Key challenges

The presence of *non-inclusive caches* makes the CRPD analysis complicated due to the *indirect effect of preemption*. The *indirect effect of preemption* is created when a particular memory reference was an *L1 cache hit* in the absence of preemption, but the same memory reference has to access the L2 cache after preemption. This counter-intuitive scenario is explained through Figure 8.2. Figure 8.2 demonstrates the indirect effect of preemption on a memory block m' which was contained *exclusively* in the L2 cache before preemption. m' was not evicted by the preempting task. However, a different memory block m, which was exclusively in the L1 cache before preemption, was evicted by the preempting task. Consider the memory access sequence $m \rightsquigarrow m'$ after the preempted task resumes execution. m will be reloaded in both the L1 and L2 cache — eventually evicting m' from the L2 cache. Therefore, even though m' was not directly evicted by the preempting task, reference to m' will suffer an additional *L2 cache miss* after preemption.

Apart from accounting the cost of *indirect preemption effect* separately, the phenomenon shown in the preceding creates several other challenges during CRPD analysis. As a result, some major changes are required in the CRPD analysis framework. Before going into the details of

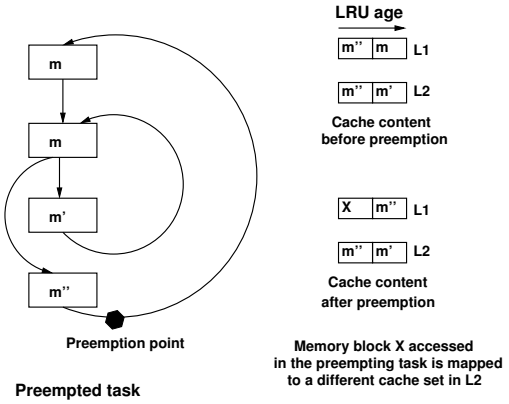


Figure 8.2: Cache reload delay due to the *indirect effect of preemption*

analysis, let us go through a few of examples, which will help understanding the main difficulties in CRPD analysis in the presence of (shared) L2 cache.

The first difficulty arises in deciding the *granularity* of component, for which the preemption cost need to be accounted. With the presence of only L1 cache and LRU cache replacement policy, total preemption cost can be computed *soundly* by accumulating the preemption cost to reload *each L1 cache block*. The *soundness* of this approach can intuitively be explained as follows: once an L1 cache block is reloaded in the cache after preemption, it can only be evicted by the *intra-task cache conflicts*. Since L1 cache is always accessed, in the presence of LRU cache replacement policy, the amount of intra-task cache conflicts does not change due to preemption. Therefore, the CRPD computation in previous literature searches only for the *next possible use* of a particular cache block after the preemption point [28; 33]. If the *next use* of a cache block $\mathcal{C}$ is an *L1 cache hit* after preemption (or an *L1 cache miss* in the absence of preemption), no preemption cost is accounted for cache block $\mathcal{C}$.

Due to the indirect effect of preemption, the amount of *intra-task cache conflicts* generated in the L2 cache *may increase* after preemption. Therefore the reasoning, as described in the preceding, may lead to underestimation in CRPD computation in the presence of L2 cache. The situation can be explained by Figure 8.3(a). Assume $m, m1, m2$ map to the same L1 and L2 cache set as shown in Figure 8.3(a). Figure 8.3(a) demonstrates a sequence of memory references $m1 \rightsquigarrow m \rightsquigarrow m1 \rightsquigarrow m2 \rightsquigarrow m$. In the absence of preemption, the L1 and L2 cache contents are shown at the left of each memory reference. The corresponding L1 and L2 cache contents are shown at the right of each memory reference after preemption. Note that the last access to m is an *L2 cache hit* in the absence of preemption, but an *L2 cache miss* after

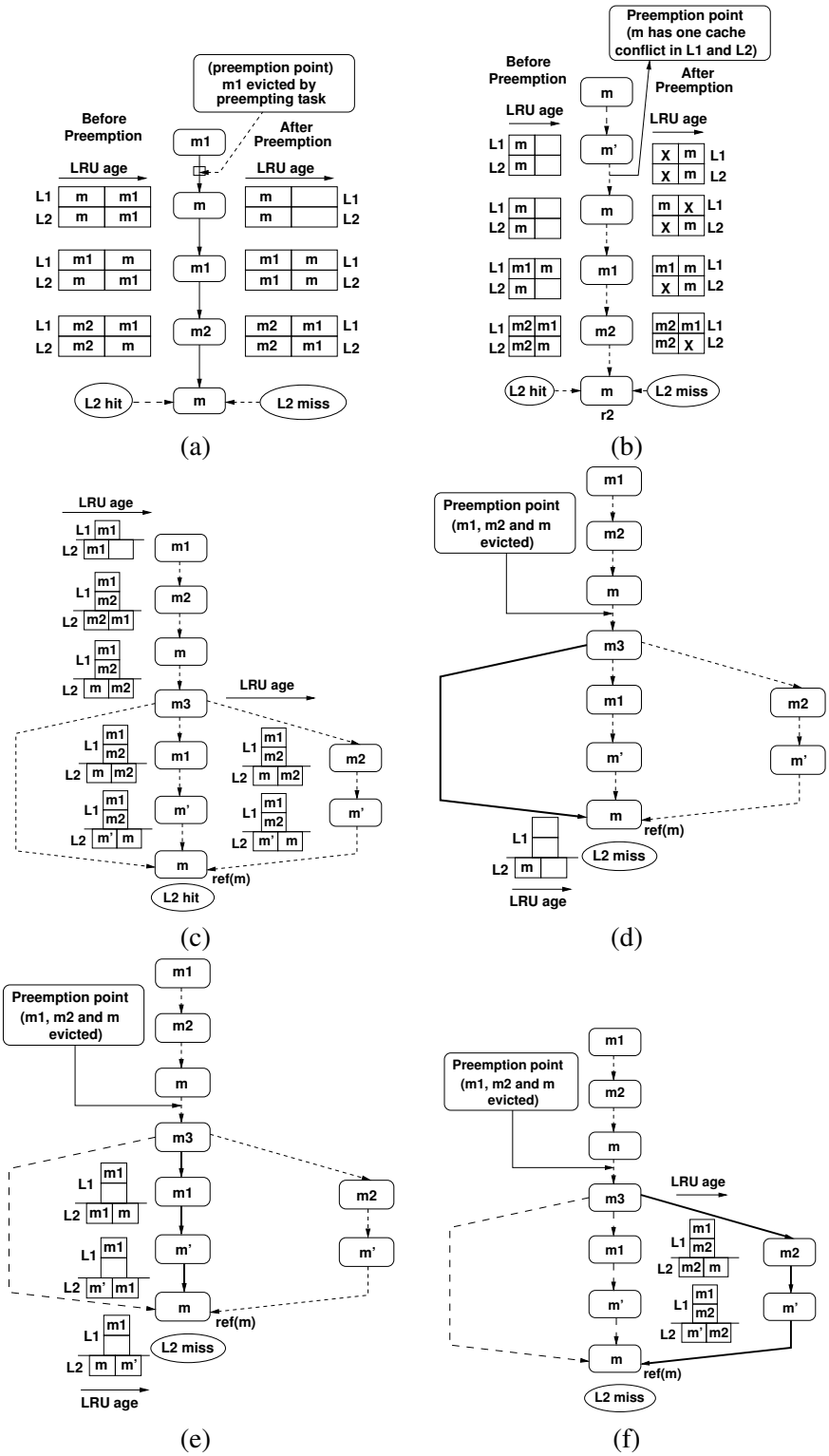


Figure 8.3: For all the figures, LRU age direction has been indicated. The direction of the arrow labelled "LRU age" points to the *older age* blocks. (a): Due to the *indirect effect* of preemption, preemption cost must go through all the memory references (not just all the memory blocks). The phenomenon is shown for memory block m. (b): In the figure, an *L2 cache miss* occurs for the second access (but first access to L2 cache) of m after preemption. (c)&(d)&(e)&(f): Demonstrating the indirect effect of preemption. (c): L1 and L2 cache contents in the absence of preemption, (d)&(e)&(f): The solid paths are the executed paths (in the order (d)→(e)→(f)) after preemption. L1 and L2 cache contents after preemption are shown when the solid path is executed.

preemption. Consider a CRPD analysis framework which is tailored for an LRU replacement policy based L1 cache. Such a CRPD analysis framework will only look till the first access of m, which is an *L1/L2 cache miss* even in the absence of preemption. Therefore, no preemption cost is added for the L1/L2 cache block corresponding to m — leading to an underestimation in the CRPD computation as shown by our example.

The example in Figure 8.3(b) shows the necessity of considering memory references (instead of memory blocks) even in the absence of *indirect effect*. Assume that $m1$ and $m2$ conflict with m in L1 cache, but only $m2$ conflicts with m in L2 cache. m' does not conflict with any of m, $m1$ or $m2$ in both the cache levels. The example shows that the second access of m after preemption ($r2$) suffers one *L2 cache miss*. This is due to the reason that L2 cache is not always accessed. Therefore, the *inter-task L2 cache conflict* (denoted by "X" in Figure 8.3(b)) is only *realized* at $r2$ (*i.e.* when the L2 cache was accessed to fetch m).

Our next example discusses the following question: *How many times a particular memory reference $ref(m)$ may suffer an L2 cache miss due to the indirect effect of preemption?* If $ref(m)$ is not accessed inside any loop, clearly, $ref(m)$ can suffer *at most one L2 cache miss* after preemption. Therefore, the more interesting scenario occurs when $ref(m)$ is accessed inside some loop.

Figure 8.3(c) shows a sequence of memory reference in the absence of preemption. For the sake of illustration, we shall assume the following:

- m and m' map to the same L1 and L2 cache set.
- $m1$ and $m2$ map to the same L2 cache set as m but m, $m1$ and $m2$ all map to different L1 cache sets.
- $m3$ is a loop header and has three different paths to $ref(m)$. $m3$ does not conflict in cache with m, m', $m1$ or $m2$.

Note that the above mapping is possible when the L1 cache has more number of *sets* than the L2 cache. Figure 8.3(c)-(f) only demonstrate a portion of L1 and L2 cache which is relevant for this discussion. For example, we do not show the mapping of m or m' in L1 cache, as it is irrelevant for our current discussion. Figure 8.3(c) clearly shows that $ref(m)$ was an *L2 cache hit* and an *L1 cache miss* in the *absence of preemption*.

Figure 8.3(d)-(f) shows the execution of three different paths reaching $ref(m)$ after the preemption. The solid line represents the executed path. After executing the path shown in

Figure 8.3(d), m is first loaded in L1 and L2 cache. Since $m1$ was evicted from L1 cache by the preempting task, it is loaded in both the L1 and L2 cache as shown in Figure 8.3(e) — generating an additional L2 cache conflict to memory block m. As a result $ref(m)$ suffers an L2 cache miss at the end. Since $m2$ was also evicted from the L1 cache, $m2$ also generates an additional L2 cache conflict to memory block m, as shown in Figure 8.3(f). Consequently, $ref(m)$ suffers a *second L2 cache miss* due to the indirect effect of preemption.

This example shows that $ref(m)$ suffers three L2 cache misses due to preemption: the first L2 cache miss (*i.e.* Figure 8.3(d)) is *directly* due to preemption, as m was evicted by the preempting task. However, the last two L2 cache misses suffered by $ref(m)$ result indirectly through two different memory blocks (i.e. $m1$ and $m2$).

It is, however, infeasible to enumerate the different paths to a particular memory reference as shown in Figure 8.3(d)-(f). Therefore, a reasonable question to ask is whether the number of L2 cache misses due to the indirect effect of preemption is *bounded*. Our work shows that this number is bounded and depends on the organization of L1 and L2 cache. More precisely, we state the following properties: Assume an L1 (L2) cache with number of cache sets S_1 (S_2) and associativity K_1 (K_2). For any memory reference $ref(m)$, assume that $IL2_{ind}$ denotes the number of *additional L2 cache misses* due to the indirect effect of preemption. We can prove the following bounds (for proofs, refer to Section 8.4):

- If $S_1 > S_2$, $IL2_{ind} \leq (\frac{S_1}{S_2})K_1 - 1$.
- If $S_1 \leq S_2 \wedge K_1 > K_2$, $IL2_{ind} \leq K_1 - K_2$.
- If $S_1 \leq S_2 \wedge K_1 \leq K_2$, $IL2_{ind} \leq 1$.

Nevertheless, the third cache organization ($S_1 \leq S_2 \wedge K_1 \leq K_2$) is the most common and is available in most deployed hardwares. Apart from bounding the number of L2 cache misses for a *realistic* cache architecture, the above properties also show that why the other cache organizations are not desirable for getting real time performance.

8.3 CRPD Analysis

In this section, we shall describe the CRPD computation in detail. We shall first show the CRPD computation for a two-level non-inclusive cache hierarchy without cache sharing (*i.e.* using the

framework described in Figure 10.2). Subsequently, we shall show the extension of our CRPD analysis framework for shared caches in multi-cores.

8.3.1 Flow Analysis

Foundation Throughout our discussion in the following, we shall assume that S_1 (S_2) denotes the number of cache sets in the L1 (L2) cache. On the other hand, K_1 (K_2) represents the associativity of the L1 (L2) cache.

CRPD computation revolves around the concept of useful cache block (UCB). A UCB is a block that *must be cached* before preemption and *may be used* later [33]. As the previous literature are based only on L1 cache, we first need to define the notion of UCB in a 2-level cache hierarchy.

Definition 8.3.1. *(Useful cache block in two-level cache) With respect to a specific preemption point p, a memory block m is characterized by a tuple (age_1, age_2) where $age_1 \in [1, K_1] \cup \{\infty\}$ and $age_2 \in [1, K_2] \cup \{\infty\}$. This characterization is defined as follows:*

- *m must be cached at p (either in L1 cache or in L2 cache or in both).*
- *m may be used at program point q that must be reached from p without m being evicted from both the L1 and L2 cache, and*
- *At program point q, the LRU age of memory block m is age_1 (age_2) in the L1 (L2) cache. If m is not cached in L1 (L2) at p or m is evicted from the L1 (L2) cache before reaching q, age_1 (age_2) is equal to ∞.*

According to the definition, m might be used from L1 or L2 cache in the absence of preemption. Therefore, the *inter-task cache interference* generated to m may lead to additional cache reload latency in the preempted task.

Example 8.3.2. *Consider the preemption point shown in Figure 8.2. Memory block m is contained exclusively in the L1 cache at its next use beyond the preemption point. On the other hand, memory block m' is contained exclusively in the L2 cache at its next use beyond the preemption point. Therefore, according to the Definition 8.3.1, we categorize memory block m and m' as follows: $m \mapsto (2, \infty)$, $m' \mapsto (\infty, 2)$.*

In the following, we shall describe two different *flow analysis*. The *backward flow analysis* computes the useful cache blocks with respect to a program point p. On the other hand, the

forward flow analysis computes the set of memory blocks which were *L1 cache hits* in the absence of preemption and are reachable to program point p. Note that an *L1 cache hit* may become an *L1 cache miss* after preemption and consequently, it may generate additional L2 cache conflict. Therefore, the forward flow analysis is particularly important while computing the *indirect effect of preemption* (as demonstrated in Figures 8.3(e)-(f)).

In the following discussion, we shall use the term *memory reference* to represent any static memory reference in the program. Note that different memory references may access the same memory block. This distinction is necessary as we show in Figure 8.3(a) that a *sound* CRPD computation may require to inspect the different references of the same memory block.

Backward flow analysis Assume that $\mathbb{M}$ represents the set of all memory blocks that could be accessed in a program. The domain of the analysis ($\mathbb{D}$) is a cartesian product of two sets as follows:

$$\mathbb{D} : \mathbb{M} \times (\mathbb{D}_c \cup \{\top\}) \tag{8.1}$$

$$\mathbb{D}_c : \{0, 1, \ldots, \mathcal{K}, \infty\} \times \{0, 1, \ldots, \mathcal{K}, \infty\} \tag{8.2}$$

where $\top$ is an additional element in the abstract domain to capture the *uncertain* information during analysis, $\mathcal{K} = \max(K_1, K_2)$ and ∞ represents numbers $\geq \mathcal{K} + 1$. $\mathbb{D}_c$ is used to capture the inclusion pattern of a memory block in the two-level cache hierarchy.

We additionally define the following function which is used throughout the discussion:

$$\Delta : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\Delta(cu_1, cu_2) = \begin{cases} 0, & \text{if } cu_1 \leq K_1; \\ LAT_1, & \text{if } cu_1 > K_1 \wedge cu_2 \leq K_2; \\ LAT_1 + LAT_2, & \text{otherwise.} \end{cases} \tag{8.3}$$

LAT_1 and LAT_2 represent the fixed L1 and L2 cache miss latencies, respectively. Therefore, Δ computes the access latency of a memory block from its given inclusion pattern in the two-level

cache hierarchy. For clarity, we shall sometimes use the notation $\Delta(cu)$ where cu will denote a tuple (cu_1, cu_2) and we shall capture the elements cu_1 and cu_2 by $cu(1)$ and $cu(2)$, respectively.

$\mathbb{D}_c$ is a *partially ordered set*. We define the partial order $\preceq$ between a pair of elements cu_1, cu_2 ($\in \mathbb{D}_c$) as follows:

$$cu_1 \preceq cu_2 \Leftrightarrow \forall ce \in \mathbb{D}_c.\ \Delta(cu_1 \boxplus ce) - \Delta(cu_1) \leq \Delta(cu_2 \boxplus ce) - \Delta(cu_2) \tag{8.4}$$

where $\boxplus$ denotes the element-wise addition operation for the tuples in $\mathbb{D}_c$. Intuitively, $cu_1 \preceq cu_2$ (*i.e.* cu_2 is partially ordered higher than cu_1) if and only if cu_2 results in *equal or more cache reload latency* compared to cu_1 in the presence of *any additional cache conflict*. However, it is possible that $cu_1 \npreceq cu_2$ and $cu_2 \npreceq cu_1$. Therefore, for the purpose of our analysis, we introduce a *join semi-lattice* $\mathbb{D}_c \cup \{\top\}$ to define the *least upper bound* operator (*i.e.* the join operator) on the respective elements. $\top$ is an element in the abstract domain such that $\forall cu \in \mathbb{D}_c.\ cu \preceq \top$. We can now define the least upper bound operator $\bigsqcup$ on the set $\mathbb{D}_c \cup \{\top\}$ as follows:

$$\bigsqcup : (\mathbb{D}_c \cup \{\top\}) \times (\mathbb{D}_c \cup \{\top\}) \rightarrow \mathbb{D}_c \cup \{\top\} \tag{8.5}$$

$$\bigsqcup(cu_1, cu_2) = \begin{cases} \top, & \text{if } cu_1 = \top \ \vee \ cu_2 = \top; \\ cu_2, & \text{if } cu_1 \preceq cu_2; \\ cu_1, & \text{if } cu_2 \preceq cu_1; \\ \top, & \text{otherwise.} \end{cases}$$

We first perform the *must cache analysis* (using [9] and [12] for L1 and L2 cache analysis) on the preempted task. As an outcome of must cache analysis, we obtain the abstract cache content at each program point. Let us assume that the tuple $MustAge_{m,p}$ captures the ***LRU ages*** of memory block m (in both the cache levels) immediately before the program point p and as computed by the *must cache analysis*. If m is not in some cache level (L1 or L2), the LRU age of m corresponding to the cache level is considered ∞. With the above definitions, we can now define the abstract *transfer* function of our backward flow analysis as follows:

$$\tau : \mathbb{D} \times \mathbb{P} \rightarrow \mathbb{D}$$

$$\tau((m, \mathcal{CU}), p) = \begin{cases} (m, \mathcal{CU}), & \text{if } m_p \neq m; \\ (m, MustAGE_{m,p}), & \text{otherwise.} \end{cases} \tag{8.6}$$

where $\mathcal{CU} \in \mathbb{D}_c$, $\mathbb{P}$ denotes the set of all program points and m_p denotes the memory block accessed at program point p. The abstract *join* operation to combine multiple abstract cache states can be defined as follows:

$$\widehat{J}_{\mathbb{D}} : 2^{\mathbb{D}} \rightarrow \mathbb{D}$$

$$\widehat{J}_{\mathbb{D}}(D) = \bigcup_{m \in \mathbb{M}} \{(m, \mathcal{CU}_m) \mid \mathcal{CU}_m = \bigsqcup_{d \in D} \mathcal{CU}_{m,d}\} \tag{8.7}$$

where $\mathcal{CU}_{m,d} = \{\mathcal{CU} \mid (m, \mathcal{CU}) \in d\}$ and $\bigsqcup$ denotes the *least upper bound* operator as described in Equation 8.6.

Our abstract domain ($\mathbb{D}$) is initialized with $(m, (\infty, \infty))$ for all the memory blocks $m \in \mathbb{M}$. At each program point, we check the accessed memory block and apply our transfer function τ as described in Equation 8.6. Since the analysis is a backward flow analysis, the abstract cache state at the exit of a basic block is computed by combining all the abstract cache states at the entry of its successors (through the join operation in Equation 8.7). The analysis terminates when a fixed-point is obtained at each program point.

Intuitively, the backward flow analysis records the next possible usage of a memory block m beyond a certain program point. Therefore, the *backward flow analysis* is used to estimate the set of *useful cache blocks* (Definition 8.3.1) at each program point.

Forward flow analysis With respect to a program point p, the forward flow analysis computes a set of memory blocks $\mathcal{M}_p$ where each $m \in \mathcal{M}_p$ satisfies the following two conditions:

- m must be accessed along one of the paths starting from the entry point of the program and ending at p. We call such references of m *reachable references* to p.
- At least one of the reachable references of m (*w.r.t.* p) *must be an L1 cache hit* in the absence of preemption.

Therefore, the abstract domain of the analysis is all possible subsets of memory blocks accessed in the program (*i.e.* $2^{\mathbb{M}}$). The abstract transfer and join operations can simply be defined as

follows:

$$\tau' : 2^{\mathbb{M}} \times \mathbb{P} \rightarrow 2^{\mathbb{M}}$$

$$\tau'(\mathcal{M}, p) = \begin{cases} \mathcal{M} \cup \{m_p\}, & \text{if } m_p \in MustACS_{p,1}; \\ \mathcal{M}, & \text{otherwise.} \end{cases} \tag{8.8}$$

where $MustACS_{p,1}$ denotes the content of L1 cache immediately before the program point p, as computed by *must cache analysis* [9]. The abstract join operation simply performs a *set union* at the control flow merge points.

The forward flow analysis starts with the *empty set* and at each program point, we apply the transfer function τ'. The abstract cache state at the entry of each basic block is computed by taking a simple set union of all the abstract cache states at the exit of its predecessors.

Analysis of the preempting task To compute the cache reload latency accurately, we need to know the set of cache blocks possibly used by the preempting task. The set of used cache blocks by the preempting task is called the *evicting cache blocks* (ECB) [28]. Since we consider general set-associative caches, for each cache set, we compute the maximum number of cache blocks used by the preempting task. ECBs can easily be computed by performing a *may cache analysis* on the preempting task (using [12]). Let us assume $MayACS_{e,1}(i)$ and $MayACS_{e,2}(i)$ denote the content of L1 and L2 abstract cache set i, respectively, at the *exit* of the preempting task and after the *may cache analysis*. For each memory block m used by the preempted task, we define a tuple $\mathcal{CE}_m$ as follows:

$$\mathcal{CE}_m = (|MayACS_{e,1}(\mathcal{S}_{m,1})|, |MayACS_{e,2}(\mathcal{S}_{m,2})|) \tag{8.9}$$

Memory block m is mapped to cache set $\mathcal{S}_{m,1}$ ($\mathcal{S}_{m,2}$) in the L1 (L2) cache. The tuple $\mathcal{CE}_m$ captures the maximum number of cache blocks accessed by the preempting task, that map to the same cache set as m (at both the cache levels). Since *may cache analysis* always computes an over-approximation of cache content [9], Equation 8.9 ensures an over-approximation on the number of used cache blocks by the preempting task.

8.3.2 Preemption delay computation

In this section, we shall show the CRPD computation using the information generated by i) *backward flow analysis*, ii) *forward flow analysis* and iii) *must cache analysis* [9; 12]. We shall assume the following terminologies:

- $\mathcal{CU}_{m,p}$: Fixed point computed by the *backward flow analysis* with respect to a memory block m and a program point p. Therefore, $\mathcal{CU}_{m,p} \in \mathbb{D}_c \cup \{\top\}$.

- $\mathcal{RS}_{ref}$: Assume memory reference ref at program point p. $\mathcal{RS}_{ref}$ captures the fixed point computed by the *forward flow analysis* with respect to p.

With the notion of $\mathcal{CU}_{m,p}$ and $\mathcal{RS}_{ref}$, we define a quantity $\mathcal{ID}_{ref,p}$ as follows:

$$\begin{aligned} \mathcal{ID}_{ref,p} = \{m \quad | \quad & m \neq *ref \wedge m \in \mathcal{RS}_{ref} \wedge \mathcal{CU}_{m,p} \neq (\infty, \infty) \\ \wedge \quad & \mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1) > K_1 \\ \wedge \quad & \mathcal{S}_{m,2} = \mathcal{S}_{*ref,2}\} \end{aligned} \tag{8.10}$$

where $*ref$ represents the memory block accessed by memory reference ref and $\mathcal{S}_{m,2}$ represents the L2 cache set in which m is mapped.

Intuitively, a memory block $m \in \mathcal{ID}_{ref,p}$ if all of the following holds:

- m must be accessed along some path starting from the entry node and ending at ref (since $m \in \mathcal{RS}_{ref}$),

- ref must be reachable from at least one reference of m which is an *L1 cache hit* in the absence of preemption (since $m \in \mathcal{RS}_{ref}$),

- m must be a *useful cache block* (Definition 8.3.1) with respect to program point p (since $\mathcal{CU}_{m,p} \neq (\infty, \infty)$), and

- m might be accessed from the L2 cache after preemption (since $\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1) > K_1$) and generate L2 cache conflict to $*ref$ (since $\mathcal{S}_{m,2} = \mathcal{S}_{*ref,2}$).

If we consider p as the preemption point, $\mathcal{ID}_{ref,p}$ captures an *over-approximation* on the set of memory blocks, which might generate additional intra-task L2 cache conflicts to $*ref$ after preemption. Therefore, in the presence of preemption by a high priority task, any intra-task

cache conflict generated to memory reference ref is either taken into account by must cache analysis or captured through the set of memory blocks in $\mathcal{ID}_{ref,p}$.

Recall from Figure 8.3(a) that a *sound* CRPD computation may require to inspect the different references of the same memory block. Therefore, in the following discussion, we shall use the term ref for any static memory reference in a program. With respect to a preemption point p, we compute the following three components:

$DCRT_{m,p,1}$ $DCRT_{m,p,1}$ computes the additional cache reload latency for memory block m when m is accessed for the *first time* after preemption (*e.g.* in Figure 8.3(a), it computes the additional cache reload latency for memory block $m1$).

$DCRT_{ref,p,2}$ If memory reference ref was an *L1 cache miss* and *L2 cache hit* in the absence of preemption, $DCRT_{ref,p,2}$ computes the additional cache reload latency when ref is first executed after preemption (in Figure 8.3(b) it computes the additional cache reload latency for $r2$).

$ICRT_{ref,p}$ If memory reference ref was an *L1 cache miss* and *L2 cache hit* in the absence of preemption, $ICRT_{ref,p}$ computes the total cache reload latency incurred due to the *indirect effect of preemption* (in Figures 8.3(e)-(f), it computes two additional *L2 cache misses* for $ref(m)$).

Note that the additional L1 cache misses due to preemption are captured by the quantity $DCRT_{m,p,1}$. Therefore, $DCRT_{ref,p,2}$ and $ICRT_{ref,p}$ need to inspect only the memory references which were *L1 cache miss* and *L2 cache hit*, in the absence of preemption. $DCRT_{m,p,1}$ and $DCRT_{ref,p,2}$ are computed as follows (in the following, we use $*ref$ to represent the memory block accessed by ref):

$$DCRT_{m,p,1} = \begin{cases} 0, \text{ if } \mathcal{CU}_{m,p} = \top \wedge \mathcal{CE}_m = (0,0); \\ \Delta(\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1),\ \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) \\ +\mathcal{ID}_{m,p}) - \Delta(\mathcal{CU}_{m,p}), \text{ if } \mathcal{CU}_{m,p} \neq \top; \\ LAT_1 + LAT_2, \text{ otherwise.} \end{cases} \tag{8.11}$$

where $\mathcal{ID}_{m,p} = \max_{ref:*ref=m} |\mathcal{ID}_{ref,p}|$. Clearly, if the cache sets used by m are unused by the preempting task (*i.e.* $\mathcal{CE}_m = (0,0)$), no additional cache reload latency is accounted. The second case in Equation 8.11 computes the additional cache reload latency by accounting the additional cache conflicts generated after preemption. Note that for L1 cache, we only consider inter-task cache conflicts (*i.e.* $\mathcal{CE}_m(1)$). However, for L2 cache, we need to consider both the *inter-task cache conflict* (*i.e.* $\mathcal{CE}_m(2)$) and additional *intra-task cache conflicts* (*i.e.* $\mathcal{ID}_{m,p}$) generated due to the preemption.

$$DCRT_{ref,p,2} = \begin{cases} 0, \text{ if } DCRT_{*ref,p,1} = LAT_1 + LAT_2 \\ \quad \vee\ \mathcal{CU}_{*ref,p} = (\infty,\infty); \\ 0, \text{ if } MustAGE_{*ref,ref}(2) + \mathcal{CE}_{*ref}(2) \\ \quad +|\mathcal{ID}_{ref,p}| \leq K_2; \\ LAT_2, \text{ otherwise.} \end{cases} \tag{8.12}$$

$MustAGE_{m,ref}(2)$ captures the *maximum LRU age* of a memory block m in the L2 cache, immediately before the reference ref (computed by *must cache analysis* [9; 12]). The first case of Equation 8.12 captures the scenario when L2 cache reload latency of $*ref$ has already been considered during the computation of $DCRT_{*ref,p,1}$. Therefore, the first case avoids double counting the *L2 cache miss latency* for the same memory block $*ref$. We combine the effect of

above two components (*i.e.* Equations 8.11-8.12) as follows:

$$DCRT_{m,p} = DCRT_{m,p,1} + \max_{ref:*ref=m} DCRT_{ref,p,2} \tag{8.13}$$

Intuitively, $DCRT_{m,p}$ captures an *upper bound* on the cache reload latency for the following two scenarios:

- Memory block m is accessed for the first time after preemption (*e.g.* $m1$ in Figure 8.3(a))
- Memory block m is accessed for the first time *from L2 cache* after preemption and the corresponding reference was an *L2 cache hit* in the absence of preemption (*e.g.* $r2$ in Figure 8.3(b)).

If m has been reloaded in the L2 cache after preemption, m can be evicted only due to the intra-task L2 cache conflicts. Note that the intra-task L2 cache conflict may increase after preemption (*indirect effect* as shown in Figures 8.3(e)-(f)). Consider a memory reference ref which was an *L1 cache miss* and *L2 cache hit* in the absence of preemption. The cache reload latency due to the *indirect effect* can be computed as follows:

$$ICRT_{ref,p} = IL2_{ind} \times \begin{cases} 0, & \text{if } MustAGE_{*ref,ref}(2) + |\mathcal{ID}_{ref,p}| \leq K_2; \\ LAT_2, & \text{otherwise.} \end{cases} \tag{8.14}$$

From the discussion in Section 8.2, we have shown that a memory reference ref can suffer more than one L2 cache misses due to the increased intra-task L2 cache conflicts (Figure 8.3(e)-(f)). The upper bound on the number of this additional L2 cache misses is given by $IL2_{ind}$ (refer to Section 8.4 for the proof).

Final CRPD computation In the preceding, we have discussed the computation of direct and indirect effect of preemption with respect to a program point p. The CRPD captures the sum of all cache reload delays maximized over the set of all program points ($\mathbb{P}$). If REF_{L2} represents all memory references that are *L1 cache misses* and *L2 cache hits* in the absence of preemption,

the final value of CRPD can be computed as follows:

$$CRPD_{final} = \max_{p \in \mathbb{P}} \left(\sum_{m \in \mathbb{M}} DCRT_{m,p} + \sum_{ref \in REF_{L2}} ICRT_{ref,p} \right) \tag{8.15}$$

8.3.3 Handling shared caches in multi-cores

In the previous section, we have described the CRPD analysis for a two-level cache hierarchy. However, we have not specifically focused on shared caches in multi-core. In the following, we shall show how our framework can easily be adapted for CRPD analysis in the presence of shared L2 caches.

Issues with shared caches In the presence of a shared cache, additional complications arise due to the *inter-core cache conflicts*. Assume t_1 and t_2 are two concurrently running tasks on two different cores. On the other hand, t' is a high priority task assigned to the same core in which t_1 is running. Therefore, t' may preempt t_1 during its execution. Inter-core conflicts from t_2 may evict memory blocks used by t_1 from the shared L2 cache, thereby reducing the number of memory blocks to consider for CRPD computation of t_1 in the presence of t'. On the other hand, inter-core conflicts from t_2 may make a memory block in t_1 *older* in the shared L2 cache set (considering LRU cache replacement policy), resulting in more opportunities for t' to evict the same memory block. Due to the above mentioned contrasting effects, CRPD may both increase and decrease due to the presence of inter-core cache conflicts.

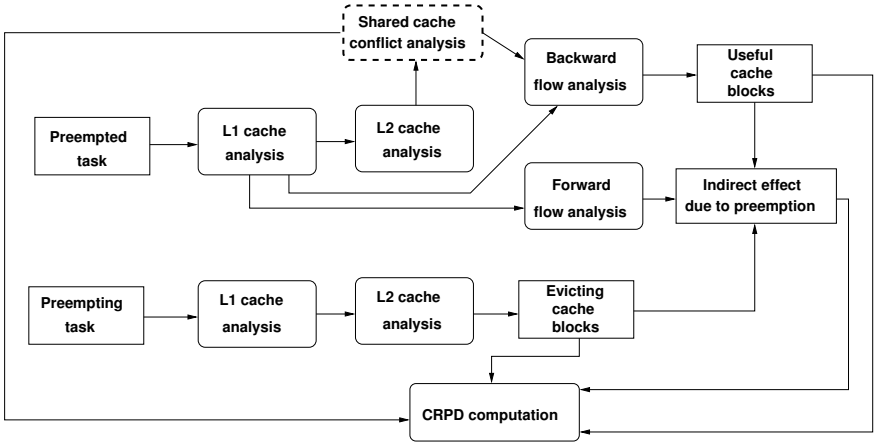


Figure 8.4: CRPD analysis framework in the presence of shared caches

Analysis framework extension Our extended framework is shown in Figure 8.4. In the previous section, we have used *must cache analysis* [9; 12] on the preempted task for computing CRPD. In the presence of shared caches, additional cache conflicts due to cache sharing are analyzed using [15]. As a result, our CRPD analysis framework uses the *must cache content* obtained after applying both [9] and [15] to deal with the shared caches. The extension due to the handling of shared caches has been highlighted by the dotted box in Figure 8.4. The *must cache analysis* using both [9] and [15] accounts the cache misses due to the *intra-task* and *inter-core* cache conflicts. Therefore, our CRPD analysis framework computes only the additional cache misses that are not accounted during *intra-task* and *inter-core* cache conflict analysis (*i.e.* during the must cache analysis using [9] and [15]).

8.4 Soundness of analysis

In this section, we shall provide the *soundness* proof of our CRPD analysis framework.

Structure of the soundness proofs

Soundness of over-approximated ECB It is always *sound* to over-approximate the set of evicting cache blocks (ECB). Recall that the CRPD computation in our framework revolves around three quantities – $DCRT_{m,p,1}$, $DCRT_{ref,p,2}$ and $ICRT_{ref,p}$ (Equations 8.11-8.14). Equations 8.11-8.14 clearly show that an over-approximation of ECB will only overestimate the value of $DCRT_{m,p,1}$, $DCRT_{ref,p,2}$ and $ICRT_{ref,p}$, keeping the overall CRPD analysis *sound*. Property 8.4.1 establishes that the set of ECBs is always *over-estimated* using our framework.

Soundness of WCET+CRPD Since CRPD analysis is normally used with WCET analysis [33], our approach guarantees a *sound* estimation of the sum of WCET and CRPD. As a result, if a memory reference is predicted *cache miss* by WCET analysis in both the L1 and L2 cache, we do not consider any additional cache miss penalty for the same memory reference during CRPD analysis. However, it is possible that the same memory reference may suffer different delays after preemption along different paths in the program. The least upper bound operator defined in Equation 8.6 ensures that we always account for the *maximum* among all possible cache reload delays. Properties 8.4.2-8.4.6 show that the *soundness* of our CRPD analysis is preserved by the partial order defined in Equation 8.4. More precisely, we show that the three key components of our CRPD computation (*i.e.* $DCRT_{m,p,1}$, $DCRT_{ref,p,2}$ and $ICRT_{ref,p}$) are

always over-estimated using the partial order defined in Equation 8.4. Finally, Property 8.4.7 ensures that we always consider a *safe* upper bound on the latency suffered by any memory reference.

Number of cache misses due to the indirect effect of preemption Recall that the presence of cache hierarchy may introduce multiple cache misses for the same memory reference after preemption. Such a scenario occurs due to the increased intra-task cache interference after preemption. We call this effect of preemption as *indirect effect*. We had introduced the bound on the number of cache misses due to the indirect effect in Section 8.2 (bound on $IL2_{ind}$). In Theorem 8.4.10, we formally prove this bound on the number of cache misses due to the indirect effect of preemption.

8.4.1 Detailed proofs

In the following, we shall discuss certain crucial properties to justify the *correctness* of our analysis. We use the following terminologies for the following discussion: S_1 (S_2) denotes the number of cache sets in the L1 (L2) cache. K_1 (K_2) denotes the associativity of the L1 (L2) cache.

As before, we shall use the term *memory reference* to capture any static memory reference in the program. Using this terminology, we can distinguish the memory references that access the same memory block.

Property 8.4.1. *Set of evicting cache blocks computed by Equation 8.9 always overestimates the actual set of evicting cache blocks in L1 and L2 cache.*

Proof. *May* cache analysis always computes an over-approximation of cache content at each program point [9; 12]. Equation 8.9 uses the L1 and L2 cache content after *may* analysis for computing $\mathcal{CE}_m$. Moreover, the may cache content is checked at the exit of the preempting task (in Equation 8.9). If a cache line is used by the preempting task in any execution, the same cache line must be used in the abstract may cache set computed at the exit of the preempting task. Consequently, the set of evicting cache blocks computed by our analysis (at both the cache levels) always over-approximates the actual set of evicting cache blocks in any concrete execution. □

Recall that we had defined a *join semi-lattice* $\mathbb{D}_c \cup \{\top\}$ and its associated partial order in

Equation 8.4. Assume that $\mathcal{CU}_{m,p}$ captures the fixed point computed by the *backward flow analysis* with respect to a memory block m and a program point p. Therefore, $\mathcal{CU}_{m,p} \in \mathbb{D}_c \cup \{\top\}$. The following discussions show that the partial order defined in Equation 8.4 preserves the *soundness* of CRPD analysis. This leads to a property that the *abstract join* operation performed during the *backward flow analysis* (Equation 8.7) does not affect the *soundness* of CRPD computation.

Property 8.4.2. *If $\mathcal{CU}_{m,p} \neq (\infty, \infty)$ and $\mathcal{CU}_{m,p} \neq \top \neq \mathcal{CU}_{m,q}$, then both of the following properties must hold when $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$:*

- $\mathcal{CU}_{m,p}(1) \leq \mathcal{CU}_{m,q}(1)$ *or* $\mathcal{CU}_{m,q}(1) > K_1$
- $\mathcal{CU}_{m,p}(2) \leq \mathcal{CU}_{m,q}(2)$ *or* $\mathcal{CU}_{m,q}(2) > K_2$.

Proof. We prove this by contradiction. Let us assume that $\mathcal{CU}_{m,q}(1) < \mathcal{CU}_{m,p}(1) \leq K_1$. We shall show that our assumption will lead to a contradiction $\mathcal{CU}_{m,p} \not\preceq \mathcal{CU}_{m,q}$.

We first construct a tuple $ce \in \mathbb{D}_c$ as follows:

- $ce = (\max(K_1 + 1 - \mathcal{CU}_{m,p}(1), 0), \max(K_2 + 1 - \mathcal{CU}_{m,p}(2), 0))$

Therefore, we have the following:

$$\mathcal{CU}_{m,p}(1) + ce(1) = \mathcal{CU}_{m,p}(1) + \max(K_1 + 1 - \mathcal{CU}_{m,p}(1), 0) \geq K_1 + 1 \tag{8.16}$$

However, as $\mathcal{CU}_{m,p}(1) > \mathcal{CU}_{m,q}(1)$, we also have:

$$\mathcal{CU}_{m,q}(1) + ce(1) = \mathcal{CU}_{m,q}(1) + \max(K_1 + 1 - \mathcal{CU}_{m,p}(1), 0) \leq K_1 \tag{8.17}$$

Since $\mathcal{CU}_{m,p}(1) + ce(1) \geq K_1 + 1$, $\Delta(\mathcal{CU}_{m,p} \boxplus ce) > LAT_1$ (recall that LAT_1 represents the fixed L1 cache miss latency). On the other hand, since $\mathcal{CU}_{m,q}(1) + ce(1) \leq K_1$, $\Delta(\mathcal{CU}_{m,p} \boxplus ce) = 0$.

According to our assumption in the beginning, $\mathcal{CU}_{m,q}(1) < \mathcal{CU}_{m,p}(1) \leq K_1$. Therefore, all of the following relationships must hold:

- $\Delta(\mathcal{CU}_{m,p}) = \Delta(\mathcal{CU}_{m,q}) = 0$,
- $\Delta(\mathcal{CU}_{m,p} \boxplus ce) - \Delta(\mathcal{CU}_{m,p}) > 0$, and
- $\Delta(\mathcal{CU}_{m,q} \boxplus ce) - \Delta(\mathcal{CU}_{m,q}) = 0$

As a result, we have $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$, but $\Delta(\mathcal{CU}_{m,p} \boxplus ce) - \Delta(\mathcal{CU}_{m,p}) > \Delta(\mathcal{CU}_{m,q} \boxplus ce) - \Delta(\mathcal{CU}_{m,q})$. This leads to a contradiction of the partial order defined in Equation 8.4.

In a similar fashion, we can assume that $\mathcal{CU}_{m,q}(2) < \mathcal{CU}_{m,p}(2) \leq K_2$ and reach a contradiction that $\mathcal{CU}_{m,p} \not\preceq \mathcal{CU}_{m,q}$. □

Recall that we use a *backward flow analysis* to compute the *useful cache blocks* (as stated in Definition 8.3.1) in the context of a two-level cache hierarchy. Intuitively, the above property ensures that we always capture the *maximum LRU age* of the respective memory blocks during the backward flow analysis.

Property 8.4.3. *If $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ holds for any memory block m, then for any memory reference ref, $\mathcal{ID}_{ref,p} \subseteq \mathcal{ID}_{ref,q}$ holds (computed by Equation 8.10).*

Proof. We prove this by contradiction. Let us assume that $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ for any memory block m, but $\mathcal{ID}_{ref,p} \nsubseteq \mathcal{ID}_{ref,q}$ for some memory reference ref. There must exist one memory block m such that $m \in \mathcal{ID}_{ref,p}$, but $m \notin \mathcal{ID}_{ref,q}$. Therefore, according to Equation 8.10, one the following conditions must hold:

- ($\mathcal{P}_1$) $\mathcal{CU}_{m,p} \neq (\infty, \infty)$, but $\mathcal{CU}_{m,q} = (\infty, \infty)$
- ($\mathcal{P}_2$) $\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1) > K_1$, but $\mathcal{CU}_{m,q}(1) + \mathcal{CE}_m(1) \leq K_1$

According to the partial order defined in Equation 8.4, we have $\forall\ cu \in \mathbb{D}_c.\ (\infty, \infty) \preceq cu$. Therefore, $\mathcal{P}_1$ reaches a contradiction, as $(\infty, \infty) \neq \mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q} = (\infty, \infty)$.

According to Property 8.4.2, $\mathcal{CU}_{m,p}(1) \leq \mathcal{CU}_{m,q}(1)$ (since $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$). Therefore, $\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1) \leq \mathcal{CU}_{m,q}(1) + \mathcal{CE}_m(1)$. This leads to a contradiction to $\mathcal{P}_2$ as mentioned in the preceding. □

Property 8.4.4. *If $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ for any memory block m, then for any memory block m, $DCRT_{m,p,1} \leq DCRT_{m,q,1}$.*

Proof. According to Property 8.4.3, if $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$, $\mathcal{ID}_{ref,p} \subseteq \mathcal{ID}_{ref,q}$. Therefore, $|\mathcal{ID}_{ref,p}| \leq |\mathcal{ID}_{ref,q}|$. Recall that

- $\mathcal{ID}_{m,p} = \max_{ref:*ref=m} |\mathcal{ID}_{ref,p}|$, and
- $\mathcal{ID}_{m,q} = \max_{ref:*ref=m} |\mathcal{ID}_{ref,q}|$.

Since, $|\mathcal{ID}_{ref,p}| \leq |\mathcal{ID}_{ref,q}|$, $\mathcal{ID}_{m,p} \leq \mathcal{ID}_{m,q}$. $\mathcal{CE}_m$ captures the number of *evicting cache blocks* mapping to the same cache set as m in L1 and L2 cache. Since $\mathcal{ID}_{m,p} \leq \mathcal{ID}_{m,q}$, we have the following two properties:

$$\mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p} \leq \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,q} \tag{8.18}$$

and using Equation 8.3, we also have

$$\begin{aligned} & \Delta(\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1), \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p}) \\ \leq \quad & \Delta(\mathcal{CU}_{m,q}(1) + \mathcal{CE}_m(1), \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,q}) \end{aligned} \tag{8.19}$$

We construct a tuple ce_a as follows:

- $ce_a = (\mathcal{CE}_m(1), \mathcal{CE}_m(2) + \mathcal{ID}_{m,q})$.

Since $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$, from the definition of partial order (Equation 8.4), we have:

$$\begin{aligned} & \Delta(\mathcal{CU}_{m,q} \boxplus ce_a) - \Delta(\mathcal{CU}_{m,q}) \\ \geq \quad & \Delta(\mathcal{CU}_{m,p} \boxplus ce_a) - \Delta(\mathcal{CU}_{m,p}) \\ = \quad & \Delta(\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1), \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,q}) - \Delta(\mathcal{CU}_{m,p}) \\ \geq \quad & \Delta(\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1), \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p}) - \Delta(\mathcal{CU}_{m,p}) \\ & \text{(using Equation 8.19)} \end{aligned} \tag{8.20}$$

Therefore, from Equation 8.11 we get $DCRT_{m,p,1} \leq DCRT_{m,q,1}$. □

Property 8.4.5. *If $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ holds for any memory block m, then for any memory block m, $DCRT_{m,p} \leq DCRT_{m,q}$ also holds.*

Proof. From Property 8.4.4, we have $DCRT_{m,p,1} \leq DCRT_{m,q,1}$ for any memory block m. Therefore, our claim can be contradicted only if the following condition holds:

- $\max\limits_{ref:*ref=m} DCRT_{ref,p,2} > \max\limits_{ref:*ref=m} DCRT_{ref,q,2}$.

According to Equation 8.12, the above condition can be satisfied if and only if one of the following conditions hold:

- ($\mathcal{P}_1$) $\mathcal{CU}_{*ref,p} \neq (\infty, \infty)$, but $\mathcal{CU}_{*ref,q} = (\infty, \infty)$.
- ($\mathcal{P}_2$) $MustAGE_{*ref,ref}(2) + \mathcal{CE}_{*ref}(2) + |\mathcal{ID}_{ref,p}| > K_2$, but $MustAGE_{*ref,ref}(2) + \mathcal{CE}_{*ref}(2) + |\mathcal{ID}_{ref,q}| \leq K_2$.
- ($\mathcal{P}_3$) $DCRT_{m,p,1} = LAT_1 + LAT_2$.

Recall that $*ref$ represents the memory block accessed by a memory reference ref and the term $MustAGE_{m,ref}(2)$ captures the *maximum LRU age* of a memory block m in the L2 cache, immediately before the reference ref (computed by *must cache analysis* [9; 15]). However, according to the partial order defined in Equation 8.4, (∞, ∞) is the *least* element of the *join semi-lattice* $\mathbb{D}_c \cup \{\top\}$. Therefore, the condition $\mathcal{P}_1$ mentioned above violates our assumption that $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ holds for any memory block m. As a result, $\mathcal{P}_1$ is infeasible.

On the other hand, from Property 8.4.3, we get $\mathcal{ID}_{ref,p} \subseteq \mathcal{ID}_{ref,q}$. Therefore, $|\mathcal{ID}_{ref,p}| \leq |\mathcal{ID}_{ref,q}|$. Consequently, we get

$$MustAGE_{*ref,ref}(2)+\mathcal{CE}_{*ref}(2)+|\mathcal{ID}_{ref,p}| \leq MustAGE_{*ref,ref}(2)+\mathcal{CE}_{*ref}(2)+|\mathcal{ID}_{ref,q}| \tag{8.21}$$

Equation 8.21 contradicts $\mathcal{P}_2$ mentioned in the preceding. Therefore, $\mathcal{P}_2$ is also infeasible.

Finally, recall that $DCRT_{m,p}$ is used to compute the maximum cache reload delay when m is accessed for the first time from L1 or L2 cache after preemption. Therefore, the maximum value of $DCRT_{m,p}$ is bounded by $LAT_1 + LAT_2$ (*i.e.* the maximum possible cache miss latency). If $DCRT_{m,q,1} = LAT_1 + LAT_2$, we have already accounted the *maximum* cost to reload the memory block m for the first time after preemption. As a result, the possibility of $\mathcal{P}_3$ also does not affect the *over-approximation* of $DCRT_{m,q}$.

Hence, $DCRT_{m,p} \leq DCRT_{m,q}$ holds for any memory block m. □

Property 8.4.6. *If $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ for any memory block m, then for any memory reference ref, $ICRT_{ref,p} \leq ICRT_{ref,q}$ (Equation 8.14).*

Proof. According to Property 8.4.3, if $\mathcal{CU}_{m,p} \preceq \mathcal{CU}_{m,q}$ holds for any memory block m, the inclusion relation $\mathcal{ID}_{ref,p} \subseteq \mathcal{ID}_{ref,q}$ holds for any memory reference ref. Therefore, $|\mathcal{ID}_{ref,p}| \leq |\mathcal{ID}_{ref,q}|$. As a result, our claim directly follows from the definition in Equation 8.14. □

Property 8.4.7. *Assume any memory reference ref in the preempted task. The latency considered for ref in our analysis always overestimates the latency suffered by ref in any concrete*

execution.

Proof. Assume m is the memory block accessed by ref. In the presence of *LRU cache replacement policy*, the amount of *inter-task cache conflicts* (generated by the preempting task) can only affect the *first reference* of a memory block m after preemption. L1 cache is always accessed. Therefore, if we want to compute the effect of preemption in the L1 cache, it is sufficient to check the first reference of m after the preemption point. For L2 cache, however, it is not sufficient to check only the first reference of m. This is due to the fact that L2 cache is not accessed if the referenced memory block is found in the L1 cache. In the following, therefore, we distinguish between the two cases:

- First memory access of block m after preemption, and
- all next accesses of block m after preemption.

Case I: m is accessed first time after preemption Let us assume $MustAGE_{m,ref} (\in \mathbb{D}_c)$ captures the maximum LRU ages of memory block m (computed by the must cache analysis using [9; 15]) in L1 and L2 cache before the memory reference ref. Therefore, our analysis framework computes the latency for memory reference ref as follows:

$$\Delta(MustAGE_{m,ref}) + DCRT_{m,p,1} \tag{8.22}$$

With respect to a program point p and a memory block m, assume that $\mathcal{CU}_{m,p}$ denotes the fixed-point computed by our *backward flow analysis*. Therefore, $\mathcal{CU}_{m,p} \in \mathbb{D}_c \cup \{\top\}$.

Due to the *join* operation (refer to Equation 8.7) performed during our *backward flow analysis*, the following partial-order relationship must hold:

$$MustAGE_{m,ref} \preceq \mathcal{CU}_{m,p} \tag{8.23}$$

Expanding Equation 8.22, we get the following value of the overall latency computed for mem-

ory reference ref:

$$\begin{aligned}
& \Delta(MustAGE_{m,ref}) + \Delta(\mathcal{CU}_{m,p}(1) + \mathcal{CE}_m(1), \\
& \mathcal{CU}_{m,p}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p}) - \Delta(\mathcal{CU}_{m,p}) \\
\geq\ & \Delta(MustAGE_{m,ref}) + \Delta(MustAGE_{m,ref}(1) + \mathcal{CE}_m(1), \\
& MustAGE_{m,ref}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p}) - \Delta(MustAGE_{m,ref}) \\
& \text{(using Equation 8.4 and Equation 8.23)} \\
=\ & \Delta(MustAGE_{m,ref}(1) + \mathcal{CE}_m(1), MustAGE_{m,ref}(2) + \mathcal{CE}_m(2) + \mathcal{ID}_{m,p}) \quad (8.24)
\end{aligned}$$

Correctness of *must* analysis ([9; 15]) ensures that $MustAGE_{m,ref}$ is always over-estimated with respect to any concrete execution. $\mathcal{CE}_m$ is computed through *may cache analysis* ([12]) and the over-estimation of $\mathcal{CE}_m$ has been discussed in Property 8.4.1. Finally, Property 8.4.3 ensures the over-estimation of $\mathcal{ID}_{m,p}$.

Therefore, the above processing always ensures an overestimation of the actual latency incurred for memory reference ref in any concrete execution.

Case II: All non-first accesses of m after preemption If ref is not the first reference of memory block m after preemption, it can face additional cache reload latency due to the following reasons:

- ($\mathcal{P}_1$) ref is the first access to L2 cache after preemption and ref was an *L2 cache hit* in the absence of preemption (*e.g.* $r2$ in Figure 8.3(b)).
- ($\mathcal{P}_2$) ref suffers an L2 cache miss due to the *indirect effect of preemption* (*e.g.* $ref(m)$ in Figure 8.3(e)-(f)).

Property 8.4.5 together with the must cache analysis ensures that we consider an upper bound on the *L2 cache misses* due to the scenario $\mathcal{P}_1$. On the other hand, Property 8.4.6, must cache analysis and the bound on $IL2_{ind}$ (refer to Theorem 8.4.10) ensure that we consider an upper bound on the number of L2 cache misses due to the indirect effect of preemption (scenario $\mathcal{P}_2$). □

The following properties formally establish the bound on $IL2_{ind}$, as discussed in Section 8.2.

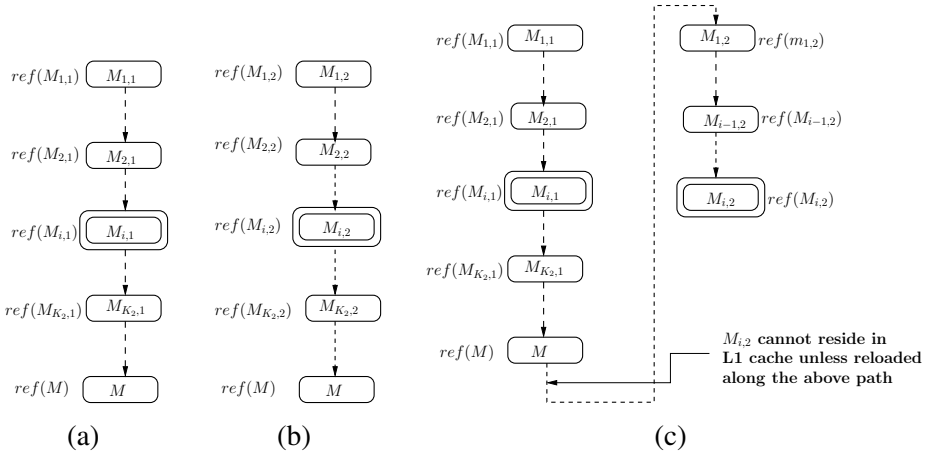


Figure 8.5: Bounding the indirect effect of preemption when $S_1 \leq S_2$ and $K_1 \leq K_2$. $ref(M_{i,1})$ and $ref(M_{i,2})$ are *L1 cache hits* in the absence of preemption, but access the L2 cache after preemption. (a)&(b): Indirect preemption effect created on $ref(M)$, (c): a scenario which shows that (a)&(b) cannot happen together

Property 8.4.8. *Assume two memory blocks m_1 and m_2 which map to the same L2 cache set. If $S_1 \leq S_2$, m_1 and m_2 map to the same L1 cache set as well.*

Property 8.4.9. *If $S_1 > S_2$, at most $(\frac{S_1}{S_2})K_1$ cache blocks in the L1 cache map to the same L2 cache set.*

Theorem 8.4.10. *Consider any memory reference $ref(M)$ in the preempted task that accesses a memory block M. Assume $ref(M)$ was an L2 cache hit and an L1 cache miss in the absence of preemption. Further assume that $ref(M)$ suffers $IL2_{ind}$ number of L2 cache misses due to the indirect effect of preemption. $IL2_{ind}$ is bounded as follows:*

- *if $S_1 \leq S_2$ and $K_1 \leq K_2$, $IL2_{ind} \leq 1$,*
- *if $S_1 \leq S_2 \wedge K_1 > K_2$, $IL2_{ind} \leq K_1 - K_2$, and*
- *if $S_1 > S_2$, $IL2_{ind} \leq (\frac{S_1}{S_2})K_1 - 1$.*

Proof. If $ref(M)$ resides outside of any loop, our claim is trivially satisfied, as $ref(M)$ can be executed at most once after the preemption. Therefore, in the following, we are concerned only about the case when $ref(M)$ is accessed within a loop.

Recall that the indirect effect of preemption may occur when some memory references access the L2 cache only after preemption, but they do not access the L2 cache in the absence of preemption (as demonstrated through Figure 8.2).

The basic idea of all the three proofs is as follows: assume that we want to impose a bound B on $IL2_{ind}$. For a memory reference $ref(M)$, we first construct B different program paths which may result in the eviction of M after the preemption — thereby generating B *level 2 cache misses* for $ref(M)$ after the preemption. If each of these B level 2 cache misses are generated due to the indirect effect of preemption, each of the B constructed path must contain at least one memory reference which access the L2 cache *only after preemption* (and not in the absence of preemption). Subsequently, we show the impossibility of constructing a $B+1$-th path (say $\mathcal{P}_{B+1}$) in a similar fashion. We shall show that any such $\mathcal{P}_{B+1}$ will contain only memory references that are either *L1 cache hit* after preemption or *L1 cache miss* even in the absence of preemption. As a result, $\mathcal{P}_{B+1}$ cannot lead to an L2 cache miss for $ref(M)$ due to the indirect effect of preemption.

$S_1 \leq S_2 \wedge K_1 \leq K_2$ If M is evicted from the L2 cache, M must have faced K_2 unique conflicts since its *last reload* into the L2 cache. Let us assume one program path $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$ (as shown by Figure 8.5(a)) which accesses K_2 unique memory blocks $\{M_{1,1}, \ldots, M_{i,1}, \ldots, M_{K_2,1}\}$ mapping to the same L2 cache set as M. If all the references in $\{ref(M_{1,1}), \ldots, ref(M_{K_2,1}\}$ access the L2 cache in the absence of preemption, M would be evicted from the L2 cache after accessing $ref(M_{K_2,1})$ even in the absence of preemption. This leads to a contradiction that $ref(M)$ is an *L2 cache hit* in the absence of preemption. Therefore, to consider the *indirect effect*, there must be one memory reference, say $ref(M_{i,1}) \in \{ref(M_{1,1}), \ldots, ref(M_{i,1}), \ldots, ref(M_{K_2,1})\}$, which does not generate L2 cache conflict in the absence of preemption (due to an *L1 cache hit*), but $ref(M_{i,1})$ generates L2 cache conflict after preemption (as $M_{i,1}$ could be evicted from the L1 cache by the preempting task).

Now consider any other program path $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$, (as shown by Figure 8.5(b)) which could create similar indirect preemption effect on memory reference $ref(M)$ after the execution of $\mathcal{P}_1$. Assume a memory reference $ref(M_{i,2}) \in \{ref(M_{1,2}), \ldots, ref(M_{i,2}), \ldots, ref(M_{K_2,2})\}$, which was an *L1 cache hit* in the absence of preemption, but will access the L2 cache after preemption (as $M_{i,2}$ can be evicted from L1 cache by the preempting task). Since L1 cache is always accessed, $ref(M_{i,2})$ can be an *L1 cache hit* (in the absence of preemption) only if the following condition holds:

- $M_{i,2}$ is accessed in the program path $\mathcal{P} := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow$

$ref(M_{K_2,1}) \rightsquigarrow ref(M) \rightsquigarrow \ldots \rightsquigarrow ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,2})$. (as shown by Figure 8.5(c)). Otherwise, $M_{i,2}$ cannot exist in the L1 cache after $\mathcal{P}_1$ is executed. This is because $K_2 \geq K_1$ and K_2 unique memory blocks $M_{1,1}, \ldots, M_{K_2,1}$ are also mapped to the same cache set as M (since $S_1 \leq S_2$). Therefore, M will be evicted from L1 cache by the set of memory blocks $M_{1,1}, \ldots, M_{K_2,1}$ after $\mathcal{P}_1$ is executed.

However, with the above condition, $M_{i,2}$ is already reloaded after preemption and before the memory reference $ref(M_{i,2})$, which makes $ref(M_{i,2})$ an *L1 cache hit* even after preemption. On the other hand, if $M_{i,2}$ is not reloaded before the memory reference $ref(M_{i,2})$, $ref(M_{i,2})$ will be an *L1 cache miss* even in the absence of preemption. Both of these scenarios lead to contradictions with our initial assumption.

$S_1 \leq S_2 \wedge K_1 > K_2$ In the above construction of $\mathcal{P}$, $M_{i,2}$ may not be evicted from the L1 cache if $K_1 > K_2$. Therefore, we first construct $K_1 - K_2$ program paths $\mathcal{P}_1, \ldots, \mathcal{P}_{K_1-K_2}$ — all leading to $ref(M)$ as follows:

- $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$
- $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$
- $\ldots$
- $\mathcal{P}_{K_1-K_2} := ref(M_{1,K_1-K_2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,K_1-K_2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,K_1-K_2}) \rightsquigarrow ref(M)$

where $\{ref(M_{i,1}), \ldots, ref(M_{i,K_1-K_2})\}$ are the set of memory references which were *L1 cache hits* in the absence of preemption but access the L2 cache after preemption. Let us construct another path, say, $\mathcal{P}_{K_1-K_2+1} := ref(M_{1,K_1-K_2+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,K_1-K_2+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,K_1-K_2+1}) \rightsquigarrow ref(M)$, where $ref(M_{i,K_1-K_2+1})$ accesses the L2 cache *only after preemption*, but not in the absence of preemption. After the execution of $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{K_1-K_2}$, there are at least K_1 unique memory block accesses in the L1 cache. Therefore, M_{i,K_1-K_2+1} is either accessed before executing the reference $ref(M_{i,K_1-K_2+1})$ (in which case, the memory reference $ref(M_{i,K_1-K_2+1})$ must be an *L1 cache hit* even after preemption), or M_{i,K_1-K_2+1} is not in the L1 cache while executing the reference $ref(M_{i,K_1-K_2+1})$ (even in the absence of preemption). In both the cases, we reach a contradiction.

$S_1 > S_2$ Assume that $B = (\frac{S_1}{S_2})K_1 - 1$. We construct $B + 1$ program paths all leading to the memory reference $ref(M)$ as follows:

- $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$
- $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$
- $\ldots$
- $\mathcal{P}_B := ref(M_{1,B}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,B}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,B}) \rightsquigarrow ref(M)$

In the above, $\{ref(M_{i,1}), ref(M_{i,2}), \ldots, ref(M_{i,B})\}$ are the set of memory references which access the L2 cache only after preemption but not in the absence of preemption.

Suppose we want to construct another program path $\mathcal{P}_{B+1} := ref(M_{1,B+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,B+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,B+1}) \rightsquigarrow ref(M)$, where $ref(M_{i,B+1})$ accesses the L2 cache *only after preemption*, but not in the absence of preemption. After $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_B$ are executed, there are at least $(\frac{S_1}{S_2})K_1$ unique memory block accesses in the L1 cache ($(\frac{S_1}{S_2})K_1$ memory blocks are $\{M_{i,1}, \ldots, M_{i,B}\} \cup \{M\}$) mapping to the same L2 cache set as memory block M. According to Property 8.4.9, there could be at most $(\frac{S_1}{S_2})K_1$ blocks in L1 cache that may map to the same L2 cache set as M. Therefore, $M_{i,B+1}$ must have been accessed along the path $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \rightsquigarrow \ldots \rightsquigarrow \mathcal{P}_B \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,B+1})$. In this case, $ref(M_{i,B+1})$ will be an L1 cache hit even after preemption. If $M_{i,B+1}$ is not accessed along the path $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \rightsquigarrow \ldots \rightsquigarrow \mathcal{P}_B \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,B+1})$, $M_{i,B+1}$ must have been evicted from the L1 cache by the set of memory blocks $\{M_{i,1}, \ldots, M_{i,B}\} \cup \{M\}$ before executing $ref(M_{i,B+1})$. As a result, $ref(M_{i,B+1})$ was an L1 cache miss even in the absence of preemption. As a result, we reach a contradiction with our assumption. □

8.5 Extension

Nested and multiple preemption

In the preceding, we have described the CRPD computation for a single preemption. Our framework can easily be extended with nested preemption. Recall that our framework computes a set of evicting cache blocks (ECB) using *may cache analysis* [12]. To handle nested preemption, we simply need to take the *union* of all the ECBs from all the higher priority tasks. More precisely, assume that $T_1, T_2, \ldots, T_n$ are the set of tasks in decreasing order of priority and we

want to compute the CRPD for T_n. A *may cache analysis* is performed for each of the tasks $T_1, T_2, \ldots, T_{n-1}$. The set of ECBs computed for $T_1, T_2, \ldots, T_{n-1}$ are then merged (*i.e.* set union) together to produce a final estimation of ECBs. Our rest of the framework remains unchanged. Since we perform a set union of all the possible ECBs, the estimated set of ECBs clearly over-approximates the set of memory blocks accessed by the set of all preempting tasks in any concrete execution.

As shown in [34], multiple preemption creates additional difficulties in the presence of *set-associative* caches. The technique proposed in [34] can be used in an exactly same fashion with our framework. For a *sound* preemption delay computation, [34] requires the set of evicting cache blocks (ECB) to be an over-approximation over *any execution* of the preempting task. Since we use *may cache analysis* to estimate the set of ECBs (as computed by Equation 8.9), we indeed over-approximate the set of ECBs over any execution of the preempting task. Secondly, the computation in [34] is based on the following insight: *if a memory block in the preempted task may be evicted by the interaction of a set of preempting tasks $T_1, T_2, \ldots, T_n$, then the same memory block may be evicted by the sequential composition $T_1 T_2 \ldots T_n$ of the set of tasks*. This insight also holds in the presence of cache hierarchy. Therefore, our framework can be used *off-the-shelf* with [34] to handle multiple preemption.

Other cache hierarchies

In this paper, we present a CRPD analysis framework for a two-level non-inclusive cache hierarchy. In multi-core architectures, inclusive cache hierarchy may limit performance when the size of the largest cache is not significantly larger than the sum of the smaller caches. Therefore, processor architects sometimes resort to non-inclusive cache hierarchies [90]. On the other hand, inclusive cache hierarchies greatly simplify the cache coherence protocol. We plan to explore inclusive cache hierarchies for CRPD computation in future.

8.6 Experimental evaluation

Experimental setup

We have chosen medium to large size benchmarks from [2], which are generally used to validate timing analysis. The code size of the benchmarks ranges from 2779 bytes (`bsort100`) to 118351 bytes (`nsichneu`), with an average code size of 18500 bytes. Throughout our evalua-

tion, we shall assume that each task has been statically mapped to a particular core and all the tasks have fixed static priorities. We compile each benchmark into simplescalar PISA (Portable Instruction Set Architecture) [81] — a MIPS like instruction set architecture. The control flow graph (CFG) of each benchmark is extracted from its PISA compliant binary and is used for all the analysis results reported here.

We choose `cnt` and `compress` from [2] to generate different amount of *inter-task* cache conflicts. `cnt` (which is a small program having a code size of 2880 bytes) is used to generate low inter-task cache conflict, whereas, `compress` (which is a relatively large program having a code size of 13411 bytes) is used to generate relatively high inter-task cache conflict. We also conduct experiments for private as well as shared L2 caches. The default micro-architectural setup is captured by Figure 10.7(a) when the L2 cache is private to each core and by Figure 10.7(b) when the L2 cache is shared among cores. For the experiments featuring a *shared L2 cache*, we use `qurt` (code size 4898 bytes) and `statemate` (code size 52618 bytes) from [2] to generate low and high inter-core cache interferences, respectively.

To validate our analysis method, the simplescalar toolset [81] was extended to support the simulation of shared L2 cache. Original simplescalar toolset supports *cycle accurate* simulation in the presence of L1 and L2 caches. However, the simplescalar toolkit does not support the simulation of shared caches in the presence of multiple cores. Such a simplescalar extension was developed in our prior work [40; 92]. The extended simplescalar framework is also *cycle accurate*. The key to such extension is to modify the main simulation loop for multiple cores. Each iteration of the main simulation loop updates the execution states on each core – mimicking the changes in execution states for each cycle on each core. As a result, the state of the shared cache is also updated appropriately for each cycle. Currently, the simulation infrastructure is limited to the simulation of homogeneous processor cores – meaning that each processor core runs at the same frequency.

As part of our work in this paper, we have extended the multi-core simplescalar developed in our prior work [40; 92] to capture the effect of preemption. We have implemented features inside the simulator by which a task can be preempted by a higher priority task and after the higher priority task finishes execution, the preempted task will resume. Before the preemption takes place, the pipeline state of the preempted task is flushed. This is acceptable, as we just want to measure the number of additional cache misses due to preemption. Such a measurement from simulation will help to evaluate the *precision* of our CRPD analysis framework. However,

it is worthwhile to mention that the search space for measuring the worst-case preemption delay is huge (as the preemption point is unknown). Therefore, the observed CRPD in our experiments may highly under-estimate the actual worst-case CRPD.

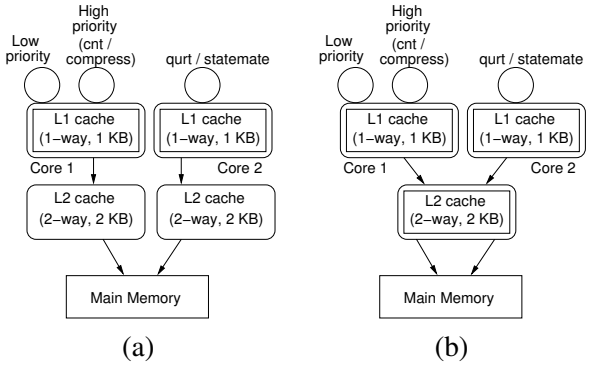


Figure 8.6: We use either `cnt` or `compress` [2] to generate inter-task cache conflict. (a) Default architecture used for the results reported as "preemption + no L2 cache sharing". (b) Default architecture used for the results using shared cache. Either `qurt` or `statemate` [2] is used to generate inter-core cache conflicts.

Our default system configuration uses a *direct-mapped*, 1 KB L1 cache and a 2-way associative, 2 KB L2 cache, both having 32 bytes cache block size. L1 cache miss penalty is 6 cycles and L2 cache miss penalty is 30 cycles.

We report the analysis overestimation ratio for the following evaluation. Overestimation ratio compares the analysis result (using our CRPD analysis framework) with the results observed from real execution (using our modified simulation infrastructure). To compare the overestimation solely due to the CRPD analysis, we record both the WCET overestimation and the overestimation of the quantity $WCET + \#p.CRPD$, where $\#p$ captures the number of preemptions. $\#p$ is chosen in a fashion so that the value of $WCET$ and the value of $\#p.CRPD$ are comparable. In the absence of preemption, we plot the WCET overestimation ratio, as the low priority task will not be interrupted by the high priority task. If the preemption is enabled, the low priority task can be preempted by the high priority task. Therefore, we record the overestimation of $WCET + \#p.CRPD$ for the low priority task. The estimation is taken using our CRPD analysis framework and Chronos WCET analysis tool [23]. The quantity $WCET + \#p.CRPD$ for a program is measured by running the same program for a few inputs, with the preemption enabled by a high priority task (as implemented in the simulation infrastructure) and recording the maximum execution time over the different inputs.

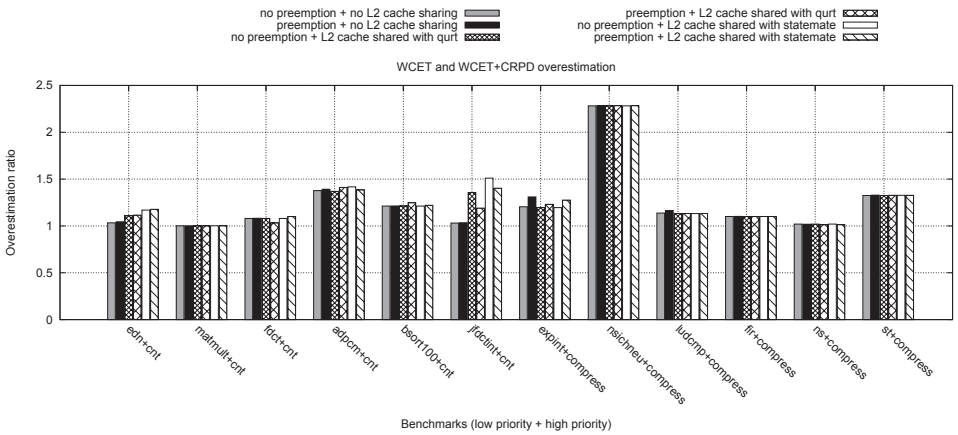


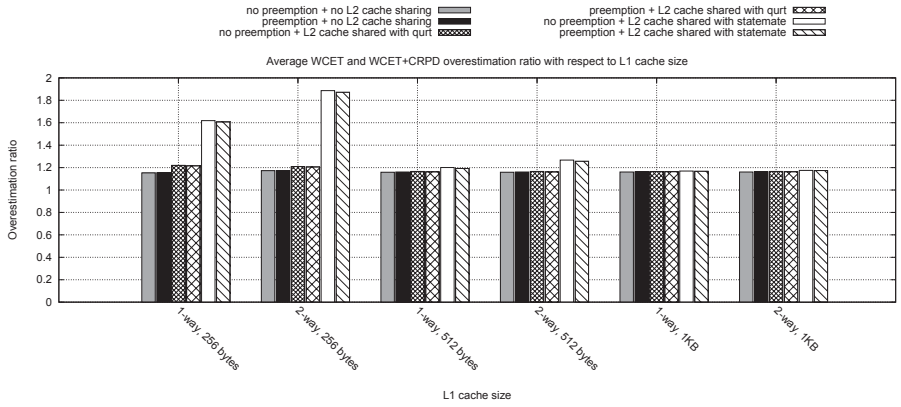
Figure 8.7: $WCET + \#p.CRPD$ overestimation for the task set used from [2]. A combination of $A + B$ along the x-axis denotes the scenario when task A is preempted by task B (where applicable)

$WCET + \#p.CRPD$ overestimation

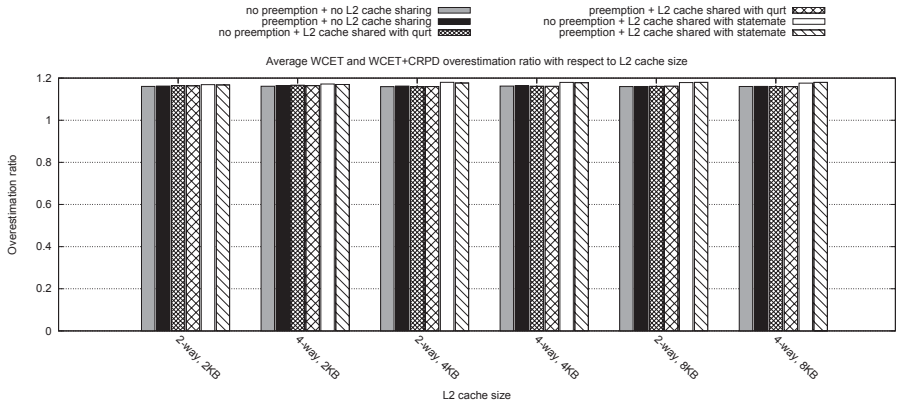
The *soundness* of our CRPD analysis is guaranteed only when used in conjunction with the WCET analysis (as motivated in [33]). Therefore, only the sum of WCET and CRPD can be compared with the measurement. Figure 8.7 shows the combined WCET and CRPD overestimation ratio in the presence of different benchmarks from [2]. Figure 8.7 clearly shows that our analysis generates *precise* estimates in most of the cases. Benchmark `nsichneu` is an exception. `nsichneu` is a benchmark with over two hundred branch instructions and many *infeasible paths*. Therefore, the overestimation largely results from the *path analysis* during the WCET computation (as evidenced by the results labeled "no preemption" in Figure 8.7).

Analysis result sensitivity w.r.t L1 and L2 cache

Figure 8.8(a) shows our analysis result sensitivity with respect to different L1 cache sizes and configurations. Similarly, Figure 8.8(b) shows the analysis result sensitivity with respect to different L2 cache sizes and configurations. Increasing the L1 cache size usually increases the number of useful cache blocks, as a bigger L1 cache can hold more cache blocks to be reused later. Consequently, CRPD may increase with bigger L1 cache, as more cache blocks can be replaced by the preempting task. However, increasing the associativity usually decreases the CRPD. This is expected, as high associativity caches reduce cache conflict misses. Therefore, the additional *inter-task* cache interferences may not be able replace some of the useful cache blocks in a 2-way associative L1 cache, as compared to a direct-mapped L1 cache of the same



(a)



(b)

Figure 8.8: CRPD and WCET analysis sensitivity with respect to (a) L1 cache configuration and (b) L2 cache configuration

size. In the same manner, increasing the L2 cache size usually increases the CRPD due to the replacement of more useful cache blocks. However, after a certain size limit of L2 cache, many useful cache blocks are not replaced due to the reduced cache interference. As a result, CRPD also decreases. Figures 8.8(a)-(b) show that our analysis is precise except for very small L1 caches in multi-cores (*e.g.* 256 bytes). This is because of the difficulties in analyzing the inter-core cache conflicts, as the overestimation also raises in the absence of preemption (refer to Figure 8.8(a)).

Effect of cache sharing

It is also worth mentioning that *measured CRPD* (using our simulation infrastructure) can be *negative* in the presence of shared caches. Since the lifetime of a task is shifted due to the preemption, it may face reduced inter-core interference after preemption. As a result, preemption of a low priority task may result in a lower number of cache misses in the shared L2 cache —

Task	Description	code size (bytes)
T1	navigation task	6496
T2	stabilisation task	2744
T3	SPI serial link control 1	1840
T4	GPS control	4048
T5	fly_by_wire servo control	1696
T6	radio control	5520
T7	SPI serial link control 2	992

Table 8.1: Papabench task set used in the evaluation

leading to a *negative* CRPD value. In our measurements, we indeed found such scenario. Nevertheless, we cannot model this scenario in our analysis, as it may require to model an unbounded number of thread interleaving patterns in concurrent programs. Therefore, the CRPD computed by our analysis is always *positive*.

Indirect effect of preemption

We have separately measured the cache reload latency due to the indirect effect of preemption (as computed by Equation (8.14)). Moreover, we have analyzed this effect for all the three different cases reported in Theorem 8.4.10 (*i.e.* for $S_1 \leq S_2 \wedge K_1 \leq K_2$, $S_1 \leq S_2 \wedge K_1 > K_2$ and $S_1 > S_2$). In general, due to the structure of the programs, the additional cache reload latency resulting from the indirect effect is *minimal*. In the worst case (over all the used benchmarks from [2]), the indirect effect of preemption is around 8% of the total CRPD cost computed by our analysis.

A case study - papabench

We have also evaluated our framework on a freely available embedded software papabench [93], a derivation from the unmanned aerial vehicle (UAV) control software Paparazzi. The controller of papabench mainly contains two modules, fly_by_wire and autopilot. fly_by_wire module is responsible for managing radio-command orders, whereas the *autopilot* module runs the navigation and stabilization tasks of the aircraft. The controller runs in two modes, namely the *manual* mode and the *automatic* mode. We evaluate our framework on the tasks shown in Table 8.1. The salient features of the tasks and their code sizes are also included in Table 8.1. We evaluate our framework for the different preemption scenarios which may appear in the real execution.

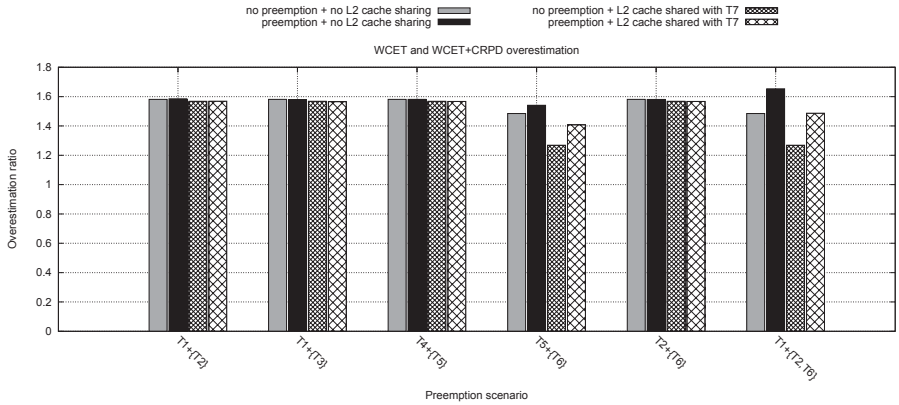


Figure 8.9: $WCET + \#p.CRPD$ overestimation for the task set used from `papabench`. A combination of $A + \{B\}$ along the x-axis denotes the scenario when task A is preempted by the set of tasks in B (where applicable)

Figure 8.9 demonstrates the combined WCET and CRPD overestimation for different preemption scenarios in `papabench`. On average, our framework generates around 55% overestimation. For nested preemptions with multiple tasks (*e.g.* preemption of $T1$ using $T2$ and $T6$ as shown in Figure 8.9), the evicting cache blocks (ECB) are merged from all the high priority tasks (*e.g.* evicting cache blocks from $T2$ and $T6$). We also observe that the overestimation in the presence of cache sharing is usually less than the same with private caches. This is mostly due to the difficulty in observing the *true worst case* of inter-task cache conflicts in the presence of preemption.

Analysis time

We have performed all the experiments in an 8-core, Intel Xeon machine with a 4 GB of RAM and running Fedora core 4 operating systems. Our analysis is fast, and finishes within a few seconds for most of the experiments. The maximum time taken by our framework is 1 minute, where we analyzed the biggest benchmarks of our test-suite (*i.e.* when we compute the CRPD for task `nsichneu` with `statemate` being run in parallel on a different core).

8.7 Chapter summary

In this chapter, we have presented a CRPD analysis framework in the presence of (shared) level two caches. We have shown that the presence of *non-inclusive caches* poses several new challenges in CRPD estimation — mainly due to the variation in *intra-task cache interferences* after preemption. We have proposed a theoretical bound on this additional *intra-task cache*

interference due to preemption and proposed a CRPD estimation framework using those bounds. Our analysis framework is *sound* and our experiments with standard WCET benchmarks as well as a real-life UAV controller application suggest that we can provide *precise* estimates for most of the cases.

Chapter 9

Modeling Cache Coherence for WCET Analysis

In this Chapter, we discuss the timing unpredictability arising due to the maintenance of cache coherence in multi-core processors. The issue of cache coherence introduces unpredictable cache coherence misses when an outdated shared data item is accessed. We propose a model to statically bound the number of coherence misses. We also show the integration of such a modeling into the existing WCET analysis framework.

9.1 Introduction

Multi-core processors introduce the problem of *cache coherence* in the presence of shared data. If a shared data item resides in the private cache of a processor core $\mathcal{C}$ and the same data item is modified by another core $\mathcal{C}'$, the data item in the private cache of processor core $\mathcal{C}$ becomes *stale*. Any access on a stale data item leads to a cache miss. A cache miss due to a stale data item is widely known in the literature as a *coherence miss*. Accurately predicting the number of coherence misses is significantly challenging. This is due to the fact that the cache coherence misses depend on the interleaving pattern of different threads running on different cores. The thread interleaving pattern is *non-deterministic* and it is, in general, infeasible to enumerate all possible thread interleaving scenario.

In this chapter, we propose to model the *cache coherence misses* for WCET analysis. Our modeling does not enumerate thread interleaving patterns and it can be applied for each core in a compositional fashion. Using the existing research on data cache analysis for single core

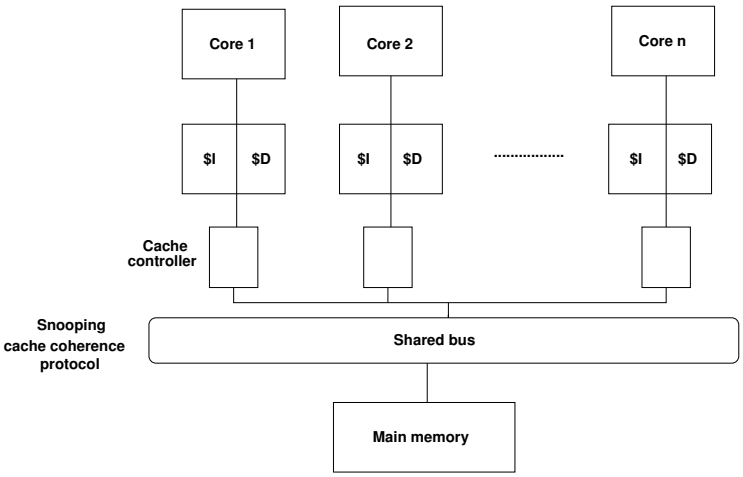


Figure 9.1: Multi-core architecture used for coherence miss modeling

[27], we first perform private data cache analysis for each core ignoring the effect of coherence misses. Subsequently, we check the shared data items that may potentially lead to stale data references. We propose the modeling of cache coherence both in the presence of *write-through* and *write-back* caches. For a *write-back* cache, we assume MESI cache coherence protocol [94]. However, our modeling of cache coherence miss can easily be adopted with minor changes in the presence of other coherence protocols. Moreover, our modeling can work also in the presence of *synchronization constructs*, which are widely used for parallel application to protect the access of a shared data item.

Our modeling of cache coherence is tightly coupled with the WCET analysis. Our goal is to predict the overall WCET of the application in the presence of shared data and cache coherence protocol. Therefore, we are only interested in the additional misses to maintain cache coherency. The data references which are analyzed as cache misses due to the intra-task cache conflicts (*e.g.* using [27]) are not considered for detecting potential coherence misses. Due to the nature of static analysis, some of the cache misses might be over-estimated while performing intra-task cache analysis using [27]. Such data references may lead to coherence misses, but they will not be considered in our coherence miss analysis (as such references are already predicted as cache misses). Consequently, our modeling of cache coherence may under-estimate the number of coherence misses in isolation, however, we guarantee the *over-approximation* of WCET using our modeling of cache coherence misses.

9.2 Overview

System and application model We assume a timing-composable multi-core architecture as shown in Figure 9.1. Each core has a private instruction and data cache. All the cores are connected to main memory through a shared bus. We do not consider *self modifying code* and therefore, we do not need to model the *coherence misses* due to the accesses of instruction memory blocks. Data can be shared among the different threads running on different cores and a cache coherence protocol guarantees the shared data coherency. We assume a well defined separation between the *private* and *shared* data section of an application. Therefore, given a memory block address, we can *statically* determine whether the memory block might be shared among different threads of the application. An address analysis mechanism can statically predict the set of memory locations accessed by a load/store instruction. However, the problem of statically predicting addresses has its own limitations. One such limitation includes the restriction on using dynamic memory allocations. Such limitations are beyond the scope of our work to address and we rely on the *precision* and *soundness* of existing address analysis techniques. However, any progress in analyzing memory locations will directly improve the precision of our coherence miss analysis.

For cache coherence, we assume a *snoopy cache coherence protocol*. Therefore, the cache controller snoops the shared bus for different bus transactions sent by different cores. We model an *invalidation* based cache coherence protocol. As a result, when a cache controller detects a *write* bus transaction on a shared data item X, it invalidates the data item X if X resides in the local cache of the cache controller. We propose the coherence miss modeling of both *write-back* and *write-through* caches. For write-through caches, a *coherence miss* may only occur for a read instruction and upon a coherence miss, the data item must be fetched from main memory. For *write-back* caches, a *coherence miss* may occur for both read and write instruction. For a *read coherence miss* in the presence of write-back cache, we assume that a successful completion of *read* operation requires at most two bus transactions - first, for flushing the modified data item into main memory and secondly, to read the same data item from main memory. On the other hand, for a *write coherence miss*, we assume that a successful completion of the write operation requires *one additional bus transaction* (which is for flushing the modified data item into main memory) to maintain cache coherency.

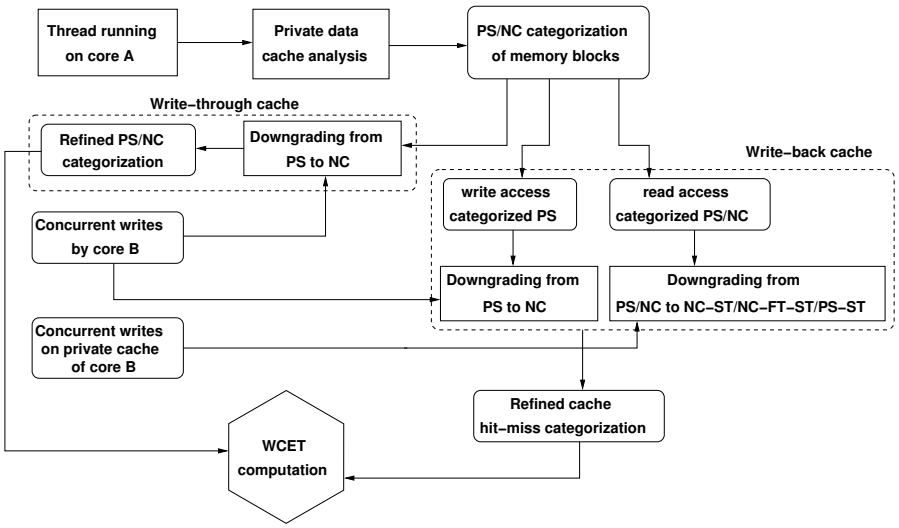


Figure 9.2: Overview of our analysis framework

Overview of analysis framework In this section, we shall give an outline of our WCET analysis framework in the presence of cache coherence misses. Figure 9.2 shows our overall analysis framework for a multi-core system with two cores (core A and core B). Assume that we want to analyze the WCET of the thread running on core A. We first perform the private data cache analysis of the program using [27] ignoring the effect of cache coherence. The private data cache analysis by [27] classifies a memory block as *persistence* (PS) or *unclassified* (NC) with respect to a program scope (*e.g.* loop nesting depth). For a *write-through* cache, the issue of cache coherence does not need to downgrade an NC categorized memory block access. This is due to the fact that an NC categorized memory access already considers both the possibilities of a *cache hit* and a *cache miss*. As a result, for a *write-through* cache, the issue of cache coherence cannot add any additional penalty for the NC categorized memory block accesses. For a *write-through* cache, we only check the PS categorized memory accesses for possible coherence misses. If there exists a concurrent write operation on a PS categorized memory block, its categorization is downgraded to NC.

For a *write-back* cache, the situation is slightly more complicated than the same in a *write-through* cache. For *write-back* caches, read and write accesses are handled separately. For write-back caches, we need to estimate the set of *dirty* and shared memory blocks in the private caches of different cores. To know whether a memory block might be modified in the private cache, we use the *may data cache analysis* proposed in [51]. As the *may data cache analysis* computes an over-approximation of data cache content, the computed cache content can be used

to find whether a write operation was performed on a *cached* memory block.

A PS categorized memory block write is downgraded to NC if there exists a concurrent write operation on the same memory block. Note that such a downgrading will consider one cache miss penalty during the WCET analysis for each such write access.

Downgrading a read access is non-trivial, as the *read* access of a memory block may suffer two cache miss penalties (one for flushing the data item into main memory and the second for reading the same data item from main memory) if the main memory is not updated or one cache miss penalty if the main memory is updated. Therefore, we consider three possible downgrading of a PS categorized memory block:

- NC : All accesses suffer at most one cache miss penalty.
- PS-ST : First access suffers at most two cache miss penalties because of a possibly *stale* data reference, but all other accesses are *cache hit*.
- NC-ST : All accesses suffer at most two cache miss penalties because each of the accesses might be a *stale* data reference.

In a similar fashion, we consider two possible downgrading of a NC categorized memory block:

- NC-FT-ST : First access suffers at most two cache miss penalties because of a possibly *stale* data reference, but all other accesses suffer at most one cache miss penalty.
- NC-ST : All accesses suffer at most two cache miss penalties because each of the accesses might be a *stale* data reference.

After all the cache access categorizations are updated, we use this cache access characterization of different memory blocks to compute the WCET of the thread running on core A.

9.3 Analysis

In this section, we shall discuss the computation of coherence cache misses in detail. Before going into the details of our analysis, we first give a brief description of the parallel programming model used for the cache analysis (in Section 9.3.1). Subsequently, we give a background on the private data cache analysis of [27] in Section 9.3.2. Our modeling of coherence cache miss is based on the work described in [27].

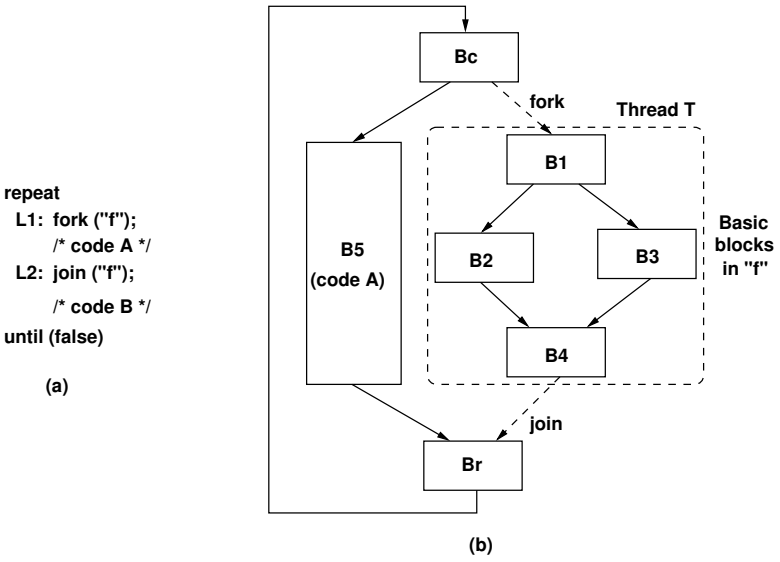


Figure 9.3: `fork` and `join` construct in a parallel program

9.3.1 Parallel programming model

We use the `fork-join` parallel programming model for our analysis. A `fork` construct creates a different thread of execution. The `fork` construct acts similar as the POSIX library function `pthread_create`. The `join` construct waits for a specific thread to terminate and the functionality of the `join` construct is similar as the library function `pthread_join` from POSIX library. Given a program with appropriate `fork` and `join` constructs, we can build the control flow graph (CFG) of the application with special control edges for distinguishing the parallel constructs (*i.e.* `fork` and `join`).

Figure 9.3(a) shows a simple example with `fork` and `join` constructs. The parent thread creates a different thread using `fork` at $L1$ and waits for the same thread to terminate before the loop iteration ends. The created thread executes a function "f" in each iteration of the loop. Note that each iteration of the loop creates as well as terminates a thread. Figure 9.3(b) shows the CFG of the program fragment shown in Figure 9.3(a) where the dotted edges capture the thread parallel constructs and $\{B1, B2, B3, B4\}$ are the set of basic blocks inside function f. The control edge $Bc \rightarrow B1$ captures a `fork` and the control edge $B4 \rightarrow Br$ captures a `join`. The set of basic blocks $\{B1, B2, B3, B4\}$ can be executed in parallel with the basic block $B5$ and therefore, may invalidate some of the shared data items used in the basic block $B5$.

9.3.2 A review of scope based data cache analysis

For private data cache analysis (which ignores the effect of cache coherence), we use the scope based data cache analysis proposed in [27]. For data cache analysis, we need to know the set of addresses accessed by each load/store instruction. Therefore, a separate address analysis phase is required which computes an upper bound on the set of addresses accessed by each load/store instruction.

In [27], a scope is defined as the loop nesting depth. A data memory reference may access different memory blocks in different iterations of a loop. For each memory block m accessed by a particular load/store instruction and for each scope, [27] defines a set of iteration interval (called as *temporal scope*) in which m could be accessed. A temporal scope $L \mapsto [x, y]$ of memory block m captures that m can *only* be accessed between iteration x and iteration y of loop L, but m can *never* be accessed *before iteration* x and *after iteration* y *of loop* L. Such a temporal scope based partitioning is quite useful for data cache analysis, as different memory blocks accessed by a load/store instruction may have totally *disjoint* temporal scopes and therefore, may not conflict in the cache with each other.

Once the temporal scopes are computed for each data reference instruction, [27] employs a scope based *persistence* analysis. Such a persistence analysis classifies each memory block accessed by a data reference as *persistence* (PS) or *unclassified* (NC) with respect to a program scope (*i.e.* loop nesting depth). Assume that a data reference instruction may access a memory block m and the instruction resides inside two loops L_1 and L_2, L_1 being the outer loop. Also assume that the scope based data cache analysis computes m as PS at $L1$, but NC at L_2. Such an output captures that m cannot be evicted from cache during a single invocation of L_2, but m might be evicted from cache outside L_2 and inside L_1.

9.3.3 Foundation

Before going into the details of cache coherence modeling, we start with a few terminologies. For a particular thread $\mathcal{T}$, let us assume that $\mathcal{F}(\mathcal{T})$ denotes the program point where the thread was created (*i.e.* the `fork` construct) and $\mathcal{J}(\mathcal{T})$ denotes the program point where the parent thread of $\mathcal{T}$ calls the `join` for thread $\mathcal{T}$. It is however possible that the parent thread never calls the `join` construct for a child thread. This specific scenario is captured by assigning a special value $\perp$ to $\mathcal{J}(\mathcal{T})$.

We distinguish between a program scope and a thread scope. Different loop nesting levels are considered different program scopes. On the other hand, a thread scope is defined as the lifetime of a thread. A thread $\mathcal{T}$ is contained inside a loop L if and only if the following condition holds:

- *$\mathcal{F}(\mathcal{T})$ and $\mathcal{J}(\mathcal{T})$ are both enclosed by L. Additionally, $\mathcal{J}(\mathcal{T})$ post-dominates $\mathcal{F}(\mathcal{T})$.*

If a thread $\mathcal{T}$ is contained inside loop L, we say that $\mathcal{T} \in resides(L)$. Note that if $\mathcal{J}(\mathcal{T}) = \perp$, $\mathcal{T} \notin resides(L)$.

Additionally, we define the following terminologies which are used throughout the rest of the chapter:

- $Par(rw)$: For any read or write operation rw in a thread $\mathcal{T}$, $Par(rw)$ denotes the set of write operations in any other thread than $\mathcal{T}$. $Par(rw)$ can be computed from the control flow graph with the added `fork-join` constructs (Figure 9.3). Each element of the set $Par(rw)$ is a tuple of the form (w, T'), where w denotes a write operation performed by T' ($\neq \mathcal{T}$).

- $\mathcal{C}(rw)$: For any read or write operation rw in a thread $\mathcal{T}$, $\mathcal{C}(rw)$ denotes the set of write operations in any thread other than $\mathcal{T}$ which may interleave among different executions of rw. Each element of the set $\mathcal{C}(rw)$ is a tuple of the form (w, T') where w denotes a write operation performed by thread T' that may execute in parallel with $\mathcal{T}$. $\mathcal{C}(rw)$ can be computed simply from the control flow graph with the added `fork` and `join` constructs as shown in Figure 9.3. Note that $\mathcal{C}(rw) \subseteq Par(rw)$.

- $\mathcal{I}_m^{rw}[L]$: For a read or write instruction rw and a loop L, $\mathcal{I}_m^{rw}[L]$ captures the range of iterations of L in which m is accessed. If m is not accessed by rw, $\mathcal{I}_m^{rw}[L] = \perp$. Note that $\mathcal{I}_m^{rw}[L]$ can be computed by the address analysis proposed in [27].

A coherence miss can be generated only for a shared memory block. Therefore, for our following discussions on coherence miss modeling, unless otherwise stated, we only consider data references that may point to shared memory region.

9.3.4 Cache coherence modeling for write-through caches

For a write-through cache, a coherence miss may be generated only for a read access. A read access on a shared data item may suffer a coherence miss if a *concurrent* write operation from a

different core can modify and therefore, invalidate the same shared data item.

Assume a read instruction which may access a memory block m and m has been categorized as *persistence* (PS) with respect to scope L after the private data cache analysis. We want to check whether accessing m may lead to a *coherence miss*. The respective PS categorization for m with respect to scope L is changed to NC if and only if the following condition holds:

$$\exists w, T: \ (w, T) \in \mathcal{C}(r) \wedge \mathcal{I}_m^w[L] \neq \perp \wedge (\mathcal{I}_m^w[L] \cap \mathcal{I}_m^r[L] \neq \phi \vee T \notin resides(L)) \tag{9.1}$$

Intuitively, the above condition captures the following scenarios:

- There is a write operation w which may execute in parallel with the thread executing r and w may access the same memory block m as also accessed by r ($(w, T) \in \mathcal{C}(r) \wedge \mathcal{I}_m^w[L] \neq \perp$).
- The write operation w and the read operation r might access the memory block m in the same iteration of L ($\mathcal{I}_m^w[L] \cap \mathcal{I}_m^r[L] \neq \phi$) or the lifetime of thread T may exceed scope L (*i.e.* $T \notin resides(L)$). Note that if thread T is contained inside loop L and additionally, r and w access m in disjoint iteration space of L, it is not possible that r will suffer any coherence miss at scope level L due to the write performed by w on memory block m.

Example 9.3.1. *Consider the CFG shown in Figure 9.3. Assume a read instruction r in basic block $B5$ and a write instruction w in basic block $B2$ may access the same memory block m. Therefore, $(w, T) \in \mathcal{C}(r)$. Assume the access of m by r is* persistent*, $\mathcal{I}_m^r[L] = [2, 5]$ and $\mathcal{I}_m^w[L] = [9, 16]$. In this case, r cannot face any additional delay for m due to coherence miss. However, if $\mathcal{I}_m^w[L] = [3, 7]$, w can invalidate the copy of m in the private cache and therefore, r might face coherence miss. As a result, the characterization of m at r will be changed to* unclassified *(NC).*

9.3.5 Cache coherence modeling for write-back caches

Write-back caches introduce additional difficulty, as the main memory might not always be updated. Assume a thread which wants to read a shared data item. If the shared data item is written by another thread in the cache, the modified data needs to be first *flushed* to main memory. Subsequently, the thread issuing the read on the same data item will fetch the data

from main memory. We enhance the cache *hit-miss* categorization of a data reference to reflect the additional scenarios arising due to cache coherence.

Cache hit-miss categorization Due to the reason mentioned in the preceding, we need the following different access categorizations of a memory block in the presence of write-back caches. All the following categorizations are scope-based, however, to keep the discussion simple, we describe the categorizations without mentioning about scopes.

1. **persistence (PS):** A PS categorized memory block can never be evicted from the cache and when the memory block is loaded for the first time, it does not have to wait for a different core to flush the data item.

2. **unclassified (NC):** A NC categorized memory block can suffer at most one cache penalty for any of its access.

3. **persistence-stale (PS-ST):** This categorization is required for a *read* access on shared data memory block. A memory block is categorized PS-ST if the first access may suffer two cache miss penalties, but all subsequent accesses lead to *cache hit*. If the data item is modified by another thread before any occurrence of the corresponding read access and the main memory is not updated, the first occurrence of the read access will suffer two cache miss penalties (for flushing the data by a remote cache controller and reading it back from main memory by the processor issuing the read). However, if no other write can be performed for the subsequent occurrences of the read and the memory block accessed by the read instruction cannot be evicted from the cache, we categorize the memory access as PS-ST.

4. **unclassified-first-stale (NC-FT-ST):** Similar to PS-ST and it is also required for the read access. A memory block is categorized NC-FT-ST if the first access may suffer two cache miss penalties but all subsequent accesses can suffer at most one cache miss penalty.

5. **unclassified-stale (NC-ST):** This categorization is required for a *read* access on shared data item. If the data item is modified by another thread and the main memory is not updated, such a read operation will require two bus transactions – first, for flushing the data into the main memory and secondly, to read the data back from the main memory. Therefore, any such read access will incur a penalty equivalent to *two cache misses*.

Downgrading cache hit-miss categorization after private data cache analysis After the private data cache analysis (*i.e.* using [27]), write accesses and the read accesses are considered separately (as shown in Figure 9.2). Recall the following scenarios for the appearance of a coherence miss:

- S_1 : A write access suffers a coherence miss. This happens if the same data is modified by another thread in its private cache without updating the main memory. In this case, the dirty memory block in the private cache of the other thread need to be flushed.
- S_2 : A read access suffers a coherence miss. In this case, the dirty memory block in the private cache of the other thread needs to be flushed and read back, suffering two cache miss penalties.

Assume a read or write instruction rw which accesses a memory block m and m has been categorized as *persistence* (PS) or *unclassified* (NC) with respect to scope L after the private data cache analysis phase. We want to check whether accessing m may lead to a *coherence miss*.

Recall that a read access may suffer two cache miss penalties due to an inconsistent main memory state. This happens when the data item is modified by another thread in the cache. A PS categorized read access can be downgraded to NC, in which case the data for the read access *might be invalid* in the private cache but the main memory *is updated*. As a result, a single memory request is sufficient to complete the read request. On the other hand, a PS categorized read access can also be downgraded to NC-ST, in which case the data for the read access *might be invalid* in the private cache and at the same time, the main memory *might also be inconsistent*. As a result, for write-back caches, we need to additionally find out the possibly *dirty* (*i.e.* modified) memory blocks which may reside in the private cache of a different thread. *A memory block might be dirty if and only if a write operation is performed on the item while cached*. Therefore, for downgrading a PS categorized memory block by a read access r to NC-ST, all of the following conditions must be satisfied:

- $\mathcal{P}_1$: There exists a write operation w in thread T such that $(w, T) \in \mathcal{C}(r)$,
- $\mathcal{P}_2$: Memory block m might be written by w, and
- $\mathcal{P}_3$: Memory block m *might be in the private cache* of T when w is performed.

Condition $\mathcal{P}_1$ can be detected by the CFG with the `fork-join` constructs and condition $\mathcal{P}_2$ can be detected by any address analysis technique. To detect the condition $\mathcal{P}_3$, we perform a may cache analysis [51] on the application. The goal of *may* data cache analysis is to over-approximate the cache content at each program point of the application. Since our goal is to get an over-approximation of the cache content, for may cache analysis, we can ignore the effect of cache coherence misses. We employ the *may data cache analysis* proposed in [51]. In the following discussion, we shall assume that $ACS_{may,rw}$ denotes the *abstract may cache content* immediately before a read/write instruction rw.

Formally, the cache access categorization of a memory block m (accessed by a read/write instruction rw) with respect to scope L is updated as follows:

- rw is a write instruction and rw is categorized PS: "PS" categorization is changed to "NC" if and and only if the following condition holds:

$$\exists w, T : (w, T) \in \mathcal{C}(rw) \wedge \mathcal{I}_m^w[L] \neq \perp \wedge (\mathcal{I}_m^w[L] \cap \mathcal{I}_m^{rw}[L] \neq \phi \vee T \notin resides(L)) \quad (9.2)$$

  The above condition is similar to *write-through* cache as the coherence miss for a write access may require at most one additional bus transaction.

- rw is a read instruction and rw is categorized PS: "PS" categorization is changed to "NC-ST" if and only if the following condition holds:

$$\begin{aligned} \exists w, T : (w, T) \in \mathcal{C}(rw) \quad & \wedge \mathcal{I}_m^w[L] \neq \perp \wedge (\mathcal{I}_m^w[L] \cap \mathcal{I}_m^{rw}[L] \neq \phi \vee T \notin resides(L)) \\ & \wedge m \in ACS_{may,w} \end{aligned} \quad (9.3)$$

  If the above condition does not hold but the condition (9.2) holds for the read access, we have $m \notin ACS_{may,w}$. Therefore, m cannot be dirty in the private cache of a different core and the main memory is updated for memory block m. Consequently, we change the "PS" categorization to "NC".

  If both the conditions (9.3) and (9.2) do not hold for the read access, "PS" categorization is changed to "PS-ST" if the following holds:

$$\exists w, T : (w, T) \in Par(rw) \wedge (w, T) \notin \mathcal{C}(rw) \wedge \mathcal{I}_m^w[L] \neq \perp \wedge m \in ACS_{may,w} \quad (9.4)$$

In this case, only the first occurrence of rw may suffer two-cache miss penalties (since the main memory may not be updated for memory block m) and rest all other accesses will be *cache hit*.

- rw is a read instruction and rw is categorized NC: The modification is similar to the modification of PS categorized memory blocks. "NC" categorization is changed to "NC-ST" if the condition (9.3) holds for the read access (since the main memory may not be updated for memory block m).

  If the condition (9.2) does not hold, then "NC" categorization is changed to "NC-FT-ST" if condition (9.4) holds (since the main memory may not updated only for the first occurrence of rw).

Putting it all together in WCET analysis After the cache access categorizations are updated, they are fed to the WCET analyzer. WCET analyzer is updated to be aware of the cache access categorizations as follows:

- **PS:** At most one cache miss penalty for the first access and cache hit for all subsequent accesses.
- **NC:** At most one cache miss penalty for each access.
- **PS-ST:** At most two cache miss penalties for first access and cache hit for all subsequent accesses.
- **NC-ST:** At most two cache miss penalties for each access.
- **NC-FT-ST:** At most two cache miss penalties for first access and at most one cache miss penalty for all subsequent accesses.

9.3.6 Cache coherence modeling in the presence of synchronization constructs

In our modeling, we have so far ignored any effect of synchronization constructs. However, in general, shared data accesses are protected by exclusive accesses in the *critical section*. The ISA of the processor usually includes atomic instructions, such as *swap*, *test&set*, *load-link-store-conditional* (LL-SC) to implement critical section. In this section, we shall show how the presence of synchronization constructs can be used to improve the analysis for predicting coherence misses.

There are mainly two reasons in which the synchronization construct information can be used to refine the value of predicted cache coherence misses:

- Synchronization constructs may create explicit ordering among different thread executions. This will refine the value of $\mathcal{C}(rw)$ for a read/write access rw, which in turn will result in a refinement of cache coherence misses.
- Critical sections protected by locks executes in a *monolithic* fashion. More precisely, no other thread can interleave the execution of a thread in the critical section, and as a result cannot generate a coherence miss.

However, the synchronization constructs usually read and write shared variables and may generate additional coherence misses during their access. In this work, we shall assume that the synchronization constructs are supported by two different instructions - `lock(x)` and `unlock(x)`, where `x` is a shared memory location.

Example 9.3.2. *Consider Figure 9.3. Assume that $B5$ has two different accesses r_1 and r_2 of a shared memory block m. Also assume that basic block $B2$ may have* concurrent *writes to m. If $B5$ is not a critical section, both r_1 and r_2 may lead to coherence misses, in the worst case, as the interleaving pattern of $B5$ and $B2$ is non-deterministic. However, if $B5$ is protected as a critical section, it executes in a monolithic fashion and only the first of r_1 or r_2 can generate a coherence miss, but not both.*

In the presence of synchronization constructs, therefore, the analysis proceeds in a similar fashion as described in previous section. However, inside a critical section, we ensure that the downgrading of cache access categorization is considered for a particular memory block only once, the very first of its access inside the critical section. The categorization for rest all other accesses of the same memory block remains unaffected.

9.4 Example

In this Section, we shall work out an example to demonstrate our analysis framework.

The example and the corresponding control flow graph (CFG) are shown in Figure 9.4. Let us assume that the set of read instructions are $\{r1, r2, r3, r4\}$ and the set of write instructions are $\{w1, w2\}$. The set of memory blocks accessed by each read/write instruction are shown beside the respective instruction in the CFG. This set of memory blocks can be found by the

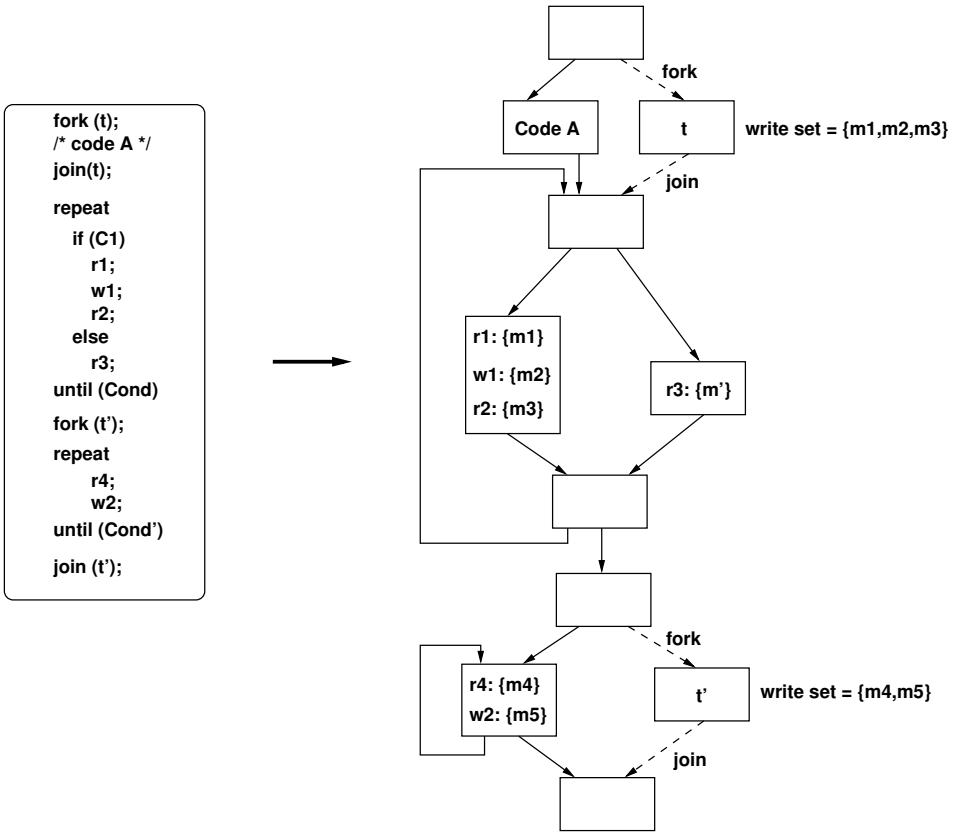


Figure 9.4: Example program and the respective control flow graph with `fork-join` constructs

address analysis technique (such as one proposed in [27]). The example program creates two threads - t and t' at different program points as shown in Figure 9.4. The code inside t and t' are not important for the discussion. However, using address analysis on t, t' we know that the set of possibly written memory blocks by t is $\{m1, m2, m3\}$ and the set of possibly written memory blocks by t' is $\{m4, m5\}$. Additionally, for the sake of illustration, we assume the following:

- We have direct mapped data cache.
- $m3$ and m' map to the same data cache set.
- $m1$, $m2$, $m3$, $m4$ and $m5$ all map to different data cache sets.

Our goal is to check the data cache access categorizations of all the read/write instructions with respect to the *global program scope*.

Write-through cache For write-through caches, we are only interested in the set of read accesses *i.e.* $\{r1, r2, r3, r4\}$. $r1$ and $r4$ will be classified as *persistence* (PS) after private data cache analysis. On the other hand, $r2$ will be classified as *unclassified* (NC), as the memory

block accessed by $r2$ (*i.e.* $m3$) might be evicted by m'. Coherence miss cannot degrade an NC categorized access for write-through caches. Therefore, we concentrate only on $r1$ and $r4$.

For $r1$, we have $\mathcal{C}(r1) = \phi$, but for $r4$, we have $\mathcal{C}(r4) = (*, t')$. Therefore, using Equation 9.1, we change the categorization of $r4$ from PS to NC. However, the categorization of $r1$ remains unchanged. Note that the lifetime of t ends before $r1$ is first executed. Therefore, the categorization of $r1$ remains *unaffected* even after our coherence miss analysis.

Write-back caches For write-back caches we need to consider the write accesses (*i.e.* $\{w1, w2\}$) and the read accesses (*i.e.* $\{r1, r2, r3, r4\}$) separately. Note that according to our assumption only $\{r2, r3\}$ are categorized as NC and reset of the accesses are categorized PS after the private data cache analysis.

For $w1$, we have $\mathcal{C}(w1) = \phi$, but for $w2$, we have $\mathcal{C}(w2) = (*, t')$. Therefore, according to Equation 9.2, the categorization of $w1$ remains unchanged, but the categorization of $w2$ is changed to NC after coherence miss analysis.

For the read accesses, we have the following:

- $\mathcal{C}(r1) = \phi$, $Par(r1) = (*, t)$.
- $\mathcal{C}(r2) = \phi$, $Par(r2) = (*, t)$.
- $\mathcal{C}(r3) = \phi$, $Par(r3) = (*, t)$.
- $\mathcal{C}(r4) = (*, t')$, $Par(r4) = \{(*, t), (*, t')\}$.

Both the conditions described in Equation 9.3 and Equation 9.2 do not hold for $r1$ and $r2$. However, the condition described in Equation 9.4 holds for both $r1$ and $r2$. Therefore, the categorization of $r1$ is changed to PS-ST (from PS) and the categorization of $r2$ is changed to NC-FT-ST (from NC) after the coherence miss analysis. Note that both $r1$ and $r2$ might face a *stale* data reference only for their *first* access. Since the memory block accessed by $r3$ is not accessed by any parallel thread, none of the conditions in Equations 9.2-9.4 are satisfied for $r3$. Therefore, the NC categorization for $r3$ remains unchanged.

Finally for $r4$, the condition described in Equation 9.3 is satisfied. Therefore, we change the categorization of $r4$ to NC-ST after our coherence miss analysis.

9.5 Chapter summary

In this Chapter, we have discussed that the timing unpredictability in multi-core may not only arise due to resource sharing, but also due to cache coherence. The challenge here lies in the modeling of cache coherence misses which occur when a core attempts to access an outdated data item from the private cache. We have proposed an analysis framework to bound the number of coherence misses. Our modeling is compositional in nature and it can easily be integrated with the state-of-the-art WCET analyzers.

Chapter 10

Static Bus Schedule aware Scratchpad Allocation in Multiprocessors

In the concluding contribution of this dissertation, we shall point to an orthogonal direction to achieve timing predictability. We shall discuss the use of our analysis framework for customized compiler optimization. Specifically, we propose a scratchpad allocation framework to reduce the shared bus traffic in multi-processors system on chip (MPSoC).

10.1 Introduction

Scratchpad memory (SPM) is a fast on-chip memory where the content of the scratchpad is controlled by the compiler and/or managed explicitly by the user. Therefore, the cost of each memory access is predictable in presence of SPM. Due to this predictability, scratchpads have been widely adopted for real-time embedded software design instead of *caches* where the memory management is entirely transparent to the user/compiler. However, explicit memory management by user is cumbersome and error-prone. Thus extensive compiler support is required for the content selection into scratchpad memories.

In this chapter, we study content selection in shared scratchpad memories for multi-processors system on chip (MPSoC) running concurrent embedded softwares. Our goal is to reduce the overall *worst case response time* (WCRT) of the application, represented as a set of task graphs. MPSoCs usually contain an on-chip scratchpad memory attached *locally* to each processing element (PE). However, a particular PE can also access other PEs' SPMs *remotely*. On the other hand, the external (off-chip) memory is accessed through a shared bus among all the available

processors in the chip.

Clearly, a processing element incurs a variable amount of delay to access the shared bus due to the bus contention introduced by other PEs. Since the requests serviced from on-chip SPMs do not access the off-chip shared bus, shared bus traffic depends on the content selection into the SPMs. On the other hand, content selection into an SPM depends on the *latency* incurred by a main memory access which in turn depends on the *waiting time* to access the shared bus. The inter-dependency between bus contention and scratchpad allocation motivates us to develop a new SPM allocation technique. Our SPM allocation method incorporates the bus schedule and hence results in a global performance optimization of the application. For the shared bus, we assume a static bus schedule using a Time Division Multiple Access (TDMA) scheme. Processors are statically assigned bus slots and the bus slots are allocated among the PEs in a round robin fashion. An integrated SPM allocation framework that considers the timing effects of shared bus in multi-processor platforms is the main contribution of our work.

To develop such an integrated SPM allocation framework for multi-processors, we face many technical challenges. Since the SPM space is shared among multiple PEs, it is important to use the shared scratchpad space as much as possible for all the *critical tasks* (*i.e.* all tasks lying in the critical path of the application) which are responsible for higher WCRT. On the other hand, if there are two processing elements PE_1, PE_2 and we fill up the shared SPM by randomly placing items from the critical tasks of PE_1, it may drastically limit the WCRT improvement of the application if the tasks running on PE_2 are also critical. Our global optimization scheme creates a unified view of all the items accessed in different processors and iteratively allocates the item(s) suffering from highest latencies to access the off-chip memory. We also employ an optimization where variables from different independent tasks may share the same SPM space through *overlay* due to their disjoint lifetimes. This leads us to more utilization of available shared SPM space.

Our allocation technique is iterative and we have used a *cycle accurate WCRT analyzer* to evaluate our approach. Our case study with real-life embedded applications such as an Unmanned Aerial Vehicle (UAV) controller and an in-orbit spacecraft software reveals that we can obtain significant WCRT reductions by appropriate content selection and *overlay* in SPM. We have also compared our approach with existing scratchpad allocation scheme which locally optimizes the per-processor execution time without being aware of variable bus delays. We have found that our approach can further improve the WCRT upto 70% compared to local scratchpad

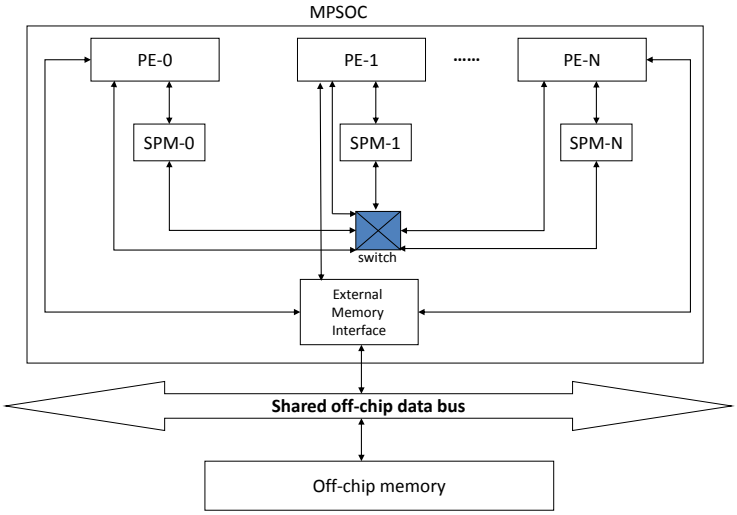


Figure 10.1: System Architecture

allocation schemes.

10.2 System and application model

In this paper, our focus is on a multi-processor architecture, as shown in Figure 10.1. The architecture contains multiple processing elements (PEs) on a chip. Each PE owns a private scratchpad memory. With respect to a specific PE, the SPMs of other PEs are referred to as remote SPMs. A PE has dedicated access to its private SPM with minimum latency. A PE can also access a remote SPM through the crossbar connecting the processors. Access to a remote SPM is relatively slower than accessing private SPM but much faster than accessing the off-chip memory. In this work, we assume that the latency to access remote SPM is bounded by a small constant (since the on-chip links generally operate on high bandwidth, this is a reasonable assumption). This kind of architecture essentially creates a *virtually shared scratchpad memory space* (VS-SPM) among all the PEs [61]. If some item is not available in VS-SPM, a processor can *bypass* the VS-SPM and fetch the memory block from slow external memory. A *bypassing* VS-SPM space creates opportunities to avoid memory spill and reloading delay as compared to its *non-bypassing* counterpart. Consequently, it leads to a fully predictable memory access behavior of the underlying application. All traffic to/from the off-chip memory has to go through a shared TDMA bus which is accessed in a round-robin fashion among all the available PEs. All on-chip SPMs are non-coherent. This helps the architecture to be free of all coherence logic required otherwise. Since the SPMs are non-coherent, there is always *at most one copy* of a particular variable in VS-SPM.

We focus here only on *scalar* and *array* variables in data memory. We assume fully separated buses and memories for both code and data. Therefore, we ignore bus traffic arising from instruction memory accesses.

We model an application as a set of task graphs where each task is mapped to exactly one PE. Each task graph is a directed acyclic graph which contains a number of tasks. Let us assume $\{T_1, \ldots, T_N\}$ be the set of N tasks corresponding to all the task graphs. A directed edge between two tasks T_x and T_y in a task graph signifies that task T_y cannot start execution before T_x finishes execution. We assume a *multi-tasking* execution model and we use a *fixed-priority preemptive* scheduling. Our goal is to derive a *compile-time* allocation of *data variables* into VS-SPM and off-chip memory to reduce the application's overall *worst case response time* (WCRT).

10.3 Overview of our SPM allocation framework

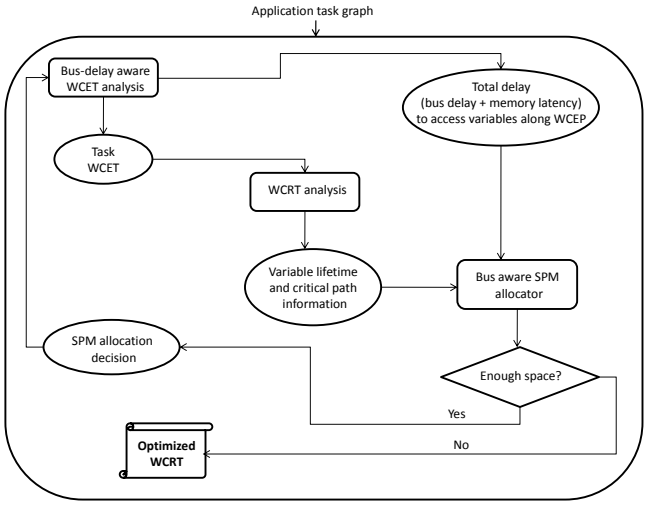
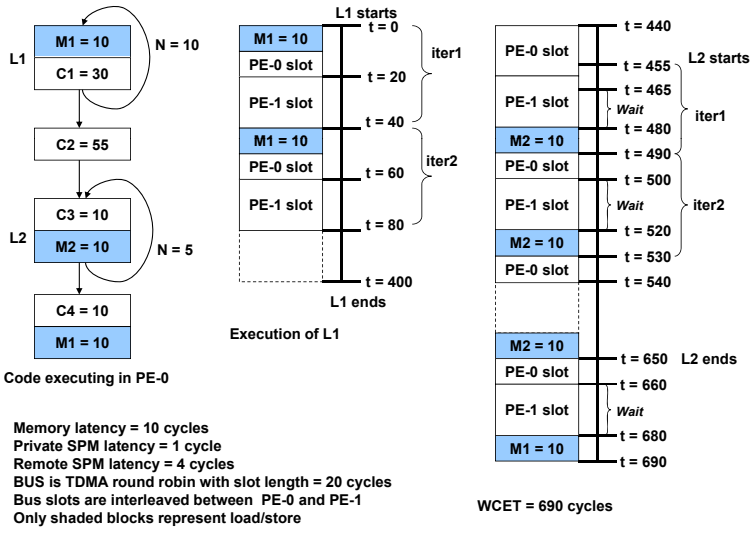
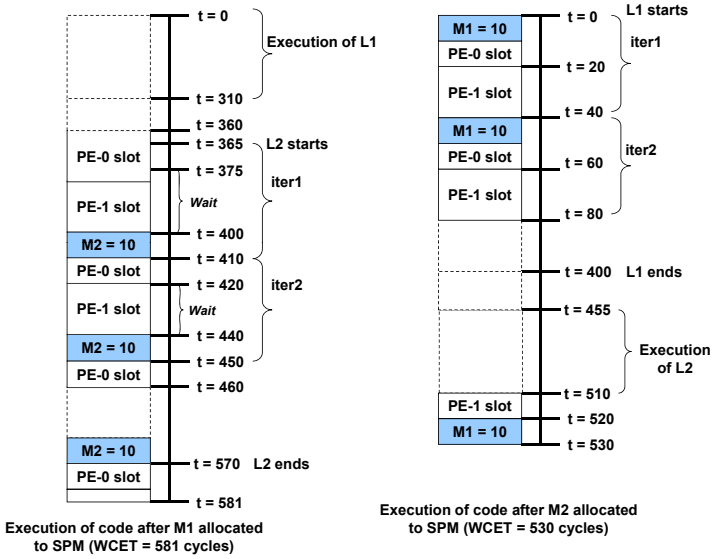


Figure 10.2: Overview of SPM allocation framework

Figure 10.2 gives a high level description of our SPM allocation framework. A *bus aware and cycle accurate* WCET analyzer computes WCETs of individual tasks together with the *external memory access profile* of each variable along the worst case execution path (WCEP). WCRT analyzer uses a fixed priority preemptive scheduling and computes the WCRT of overall application from individual WCETs of all tasks. As a by-product, the WCRT analyzer also produces the *lifetime* of each variable (the time interval between which a particular variable might be accessed) and critical path of the application. The SPM allocator computes a set of allocation decisions depending on the memory access profile of each variable and the critical path of the



(a)



(b)

Figure 10.3: (a) A sample code and its execution without SPM allocation (b) Execution of the code by two possible SPM allocations

application. An allocation decision could be either to allocate some variable in shared SPM or to revoke a previous allocation decision (i.e., to reclaim the space from a previous allocation decision and deallocate the corresponding variable(s) from SPM). Since a set of allocation decisions might change the memory access statistics and the critical path, the critical path is re-computed to produce a further set of allocation decisions.

It is important to note from Figure 10.2 that the only information flow from the bus aware WCET analysis to our SPM allocator is in the form of an external memory access profile along WCEP. The nature of shared bus is entirely hidden to the SPM allocator. Therefore, our SPM allocator is independent of the nature of shared bus used by the underlying architecture. A bus-delay aware SPM allocator is the primary focus of this work — we shall give the motivation

behind this now and discuss it further in Section 10.5.

An example Figure 10.3(a) shows a sample code and its execution at PE-0 in presence of shared bus. "C" blocks in the control flow graph (CFG) represent computations without external memory access. The number inside each block corresponds to the fixed cost of the computation. Only shaded blocks (marked with "M") in the CFG represent external memory accesses and hence might suffer from variable bus delays. We assume an external memory latency of 10 cycles and TDMA bus slot length is 20 cycles. Let us first examine the execution patterns of two loops (L1 and L2) when there is no scratchpad. Last two parts in Figure 10.3(a) demonstrate the execution behaviors of L1 and L2. We observe that references to M2 frequently suffer additional bus delays to access the off-chip memory. On the other hand, references to M1 hardly suffer any additional bus delay due to a perfect alignment with corresponding bus slots most of the time. Consequently, final WCET of the example program turns out to be 690 cycles.

We now consider an architecture with scratchpad memory (SPM). Let us assume that private SPM latency is 1 cycle and remote SPM latency is 4 cycles. For simplicity, we assume that we can allocate either M1 or M2 in the SPM but not both due to space constraints. A *bus-unaware* greedy SPM allocation (*e.g.* in [57]) scheme allocates variables to SPM by traversing them in decreasing order of their access frequencies. Since the access frequency of M1 (11) is higher than that of M2 (5), a greedy bus-unaware SPM allocator will pick M1 as the potential candidate to be allocated in the SPM. The modified execution flow is shown in the first part of Figure 10.3(b). Even though loop L1 can now be completed in fewer cycles, references to M2 still suffer high bus delays. This leads to an optimized WCET of 581 cycles.

Now assume that we allocate M2 instead of M1 in the SPM (second part of Figure 10.3(b)). Since M1 accesses are aligned to the beginning of bus slots, they will not encounter any additional bus delay as before. On the other hand, since M2 now has been allocated to SPM, its references no more encounter any bus delay. This leads to a better optimized WCET of 530 cycles.

A *bus-unaware* SPM allocation algorithm does not take into account the bus delay encountered for memory accesses. In Figure 10.3, accesses of M1 inside loop L1 do not encounter any bus delay. However, each access of M2 suffers additional bus delay. Careful examination through Figure 10.3(a) reveals that references to M2 contribute more towards the program's WCET than the references to M1. Therefore, compared to M1, M2 is a better candidate for

SPM allocation in this example.

We shall illustrate the work-flow of our iterative SPM allocation framework by using Figure 10.4. Here we shall take two tasks T1 and T2 executing concurrently at PE-0 and PE-1 respectively (Figure 10.4(a)). Task T1 is the same program as shown in Figure 10.3(a). We introduce a new task T2 as shown in Figure 10.4(a). A careful illustration similar to Figure 10.3(a) reveals that WCET of T2 is 480 cycles. Both the PEs have private SPMs (SPM-0 and SPM-1). We assume both the tasks start execution at time 0. Our goal is to minimize the overall WCRT of the application containing tasks T1 and T2.

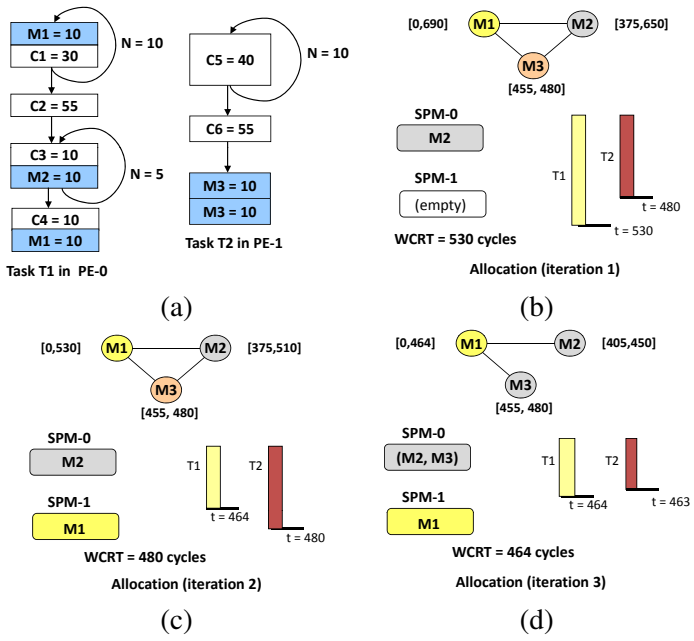


Figure 10.4: Iterative SPM allocation scheme shown on two tasks T1, T2 running on different processors. Task T1 is same as the example in Fig. 10.3.

Our technique exploits the lifetime of variable access to efficiently use the shared SPM space. Variable lifetime is indicated by an interval as shown in Figures 10.4(b)-(d). The interval represents the time span from the *earliest time* the variable could possibly be accessed to the *latest time* it could possibly be accessed. We construct an interference graph from these intervals and produce a coloring of the graph. Each individual color represents a group of variables which are accessed at disjoint time intervals. Interference graph is a globally unified graph which considers all the variables accessed in different tasks running at different PEs. In very first iteration of allocation (Figure 10.4(b)), we observe that the interference graph is a complete graph. We choose M2 to allocate in SPM-0 as task T1 is in the critical path (having larger WCET than T2) and previously, we observed that M2 suffers more memory latency to access than M1.

After allocation of M1 into SPM-0, WCRT reduces (becomes 530 cycles) but the critical path does not change (i.e., task T1) and the interference graph still remains to be a complete graph (Figure 10.4(c)). Therefore, our choice was to allocate M1 (accessed in task T1) into SPM-1 instead of M3 (accessed in task T2). This leads to a reduced WCRT of 480 cycles and we also observe that the critical path has switched to task T2 (Figure 10.4(c)). More importantly, the interference graph is no longer a complete graph as M2 and M3 are being accessed at disjoint time interval (Figure 10.4(d)). Consequently, M2 and M3 can share the same space in SPM-0 and we allocate M3 too in SPM-0 (Figure 10.4(d)). After this final allocation, WCRT further reduces to 464 cycles (recall that remote SPM latency was assumed to be 4 cycles).

Now assume the presence of a bus-unaware SPM allocator which locally optimizes per-processor execution time (as described in [63]). It would have allocated M1 in SPM-0 (for M1 having higher access frequency than M2) and M3 in SPM-1 (as remote SPM allocation is not considered), resulting in a final WCRT of 581 cycles. This example demonstrates the effectiveness of our approach, as we can considerably improve the WCRT (464 cycles) compared to [63].

10.4 Bus aware WCRT analysis

For bus-aware SPM allocation, we need to first perform a bus-aware WCET analysis of each individual task. We use our previous work on bus-aware, cycle accurate WCET analysis in [40] for the allocation framework.

The outcome of a single task analysis is a metric C_v attached to each variable v accessed in the task. C_v represents the total contribution of variable v towards WCRT of the task. This contribution includes the total waiting time to access the shared bus as well as the total off-chip memory latency for all references of variable v in the worst-case path. A TDMA bus scheduling policy is used where a bus slot is interleaved among all the PEs in a round-robin fashion. For rest of the discussion, we shall assume that there are a total $\mathcal{J}$ number of processors and the bus slot length assigned to each processor is S_l.

Computation of C_v A bus aware analysis computes the bus delay for each memory reference that may potentially access the shared bus. However, precise computation of this bus delay requires a virtual unrolling of all the loops. Our previous work in [40] uses an approximation to align the start of each loop iteration at the beginning of a new bus schedule. The alignment

avoids the virtual unrolling but it requires additional alignment cost for each loop iteration and this cost is included in the WCET computation. Artificial alignment of a loop iteration is not necessary if the loop does not contain any memory reference that may access the shared bus. Let us denote the average alignment cost of a single iteration of loop lp by Δ^{lp}. To use the analysis for bus aware SPM allocation, let us assume that $freq^{mem}$ represents the frequency of an off-chip memory reference mem along the worst case execution path (WCEP) and ∂^{mem} is the bus delay computed by the analysis for memory reference mem. Further assume that $MEM(v)$ returns the set of all off-chip memory references of variable v and $LP(mem)$ returns the immediately enclosing loop of memory reference mem. Given the above, we compute C_v as follows:

$$C_v = \sum_{mem \in MEM(v)} (\partial^{mem} + LAT + \Delta^{LP(mem)}) \times freq^{mem} \tag{10.1}$$

LAT represents the off-chip memory latency. Note that $\Delta^{LP(mem)}$ will disappear after all the variables accessed inside $LP(mem)$ are allocated in SPM. Therefore, variables incurring high memory latency and accessed inside a loop with high alignment penalty are preferred for SPM allocation. Consequently, $\Delta^{LP(mem)}$ is added as a component for computing C_v. It is interesting to notice that the SPM allocator takes only the value of C_v as input. Therefore, a more accurate analysis for computing C_v might easily improve the result generated by our SPM allocator.

Lifetime of a task WCET analysis is carried out initially and after each iteration of SPM allocation algorithm. Let us assume $wcet(t_i, A)$ denotes the WCET of task t_i under allocation A. For lifetime computation, we assign four parameters to each task as follows: $eStart(t_i, A)$ (*earliest* start time), $eFinish(t_i, A)$ (*earliest* finish time), $lStart(t_i, A)$ (*latest* start time) and $lFinish(t_i, A)$ (*latest* finish time). Given an allocation A and the corresponding value of $wcet(t_i, A)$, we can estimate the lifetime of task t_i, defined as the interval between the lower bound on the start time and the upper bound on the finish time of t_i. Therefore, lifetime of a task is captured by the interval $[eStart(t_i, A), lFinish(t_i, A)]$. This estimation takes into account the dependencies among the tasks (partial ordering imposed by the task graph) as well as preemptions. WCRT of the whole application containing N tasks under allocation A is thus given

by the following equation:

$$WCRT_{final} = \max_{1 \leq i \leq N} lFinish(t_i, A) - \min_{1 \leq i \leq N} eStart(t_i, A) \tag{10.2}$$

WCRT analysis of a single task We consider a fixed-priority preemptive scheduling. Therefore, we need to consider the preemption cost of task t_i. An application is periodic in nature. An application is modeled as a task graph and one activation of the application is the completion of this entire task graph. Therefore, all tasks in the task graph have a common period and deadline which is the period of the entire application. We denote the priority of a task t_i by $pr(t_i)$. Lower numbers are considered to be higher priority. The assigned PE to a task t_i is denoted by $PE(t_i)$. Assume that the set of tasks which may preempt task t_i is denoted by $hp(t_i)$. $hp(t_i)$ is defined as follows:

$$\begin{aligned} hp(t_i) = \quad & \{t_j \mid t_i \notin \mathcal{D}(t_j) \wedge t_j \notin \mathcal{D}(t_i) \wedge PE(t_j) = PE(t_i) \wedge \\ & pr(t_j) < pr(t_i) \wedge [eStart(t_j, A), lFinish(t_j, A)] \\ & \cap [eStart(t_i, A), lFinish(t_i, A)] \neq \phi\} \end{aligned} \tag{10.3}$$

$\mathcal{D}(t_i)$ denotes the set of tasks which depend (directly or indirectly) on task t_i according to the partial order imposed by the task graph. Therefore, $hp(t_i)$ denotes all higher priority tasks whose lifetimes may overlap with that of t_i in the same PE. WCRT of the task t_i is then computed by the following:

$$\begin{aligned} wcrt(t_i, A) = \quad & wcet(t_i, A) + \sum_{t_j \in hp(t_i)} wcet(t_j, A) \\ & + |hp(t_i)| \times \mathcal{J} \times S_l \end{aligned} \tag{10.4}$$

Since each task has the same period and deadline, a higher priority task can preempt a lower priority task executing in the same PE at most once. Preemption of a lower priority task will also disturb the external memory access profile of the preempted task beyond the preemption point, which may lead to additional bus delay. Consequently, the delay encountered for a preemption

can be at most the worst case execution time of the preempting task together with any additional bus delay encountered for preemption. Note that $|hp(t_i)| \times \mathcal{J} \times S_l$ bounds the additional bus delay. We ignore the operating system overhead due to context switch. Nevertheless, an upper bound on the context switch cost can easily be accounted during the WCRT computation of t_i ($wcrt(t_i, A)$).

We have for each task t_i: $lFinish(t_i, A) = lStart(t_i, A) + wcrt(t_i, A)$. Further, the partial ordering of tasks in the task graph imposes the constraint that a task t_i can start execution only after all its predecessors have completed execution. In other words, $lStart(t_i, A) \geq lFinish(u, A)$ for all tasks u preceding t_i in the partial order imposed by the application task graph.

For WCRT analysis, we also need to compute the best case execution time (BCET) of each task t_i. BCET of t_i under allocation A is denoted by $bcet(t_i, A)$. We use the following for BCET computation: (i) unless a variable is already allocated to some remote SPM, its location is considered to be the private SPM (i.e., no external memory access is considered when computing $bcet(t_i, A)$). (ii) no preemption cost needs to be considered for BCET (the best-case scenario). Therefore, for each task t_i: $eFinish(t_i, A) = eStart(t_i, A) + bcet(t_i, A)$. Further, due to the partial ordering of tasks in the task graph, $eStart(t_i, A) \geq eFinish(u, A)$ for all tasks u preceding t_i in the partial order imposed by the application task graph.

10.5 Bus-delay aware Scratchpad allocation

In this section, we describe our iterative SPM allocation algorithm in details. An optimal solution in our setting is clearly *infeasible*. In presence of Q processing elements in the MPSoC, each variable has $Q + 1$ possible places to reside (one in each SPM and the external memory). Consequently, an exhaustive search requires to explore $(Q + 1)^n$ possibilities with n variables, which is clearly infeasible even if n is relatively small. Therefore, in the following discussion, we propose an iterative heuristic which computes a solution very fast and still overpowers previously proposed local scratchpad allocation schemes.

In the following discussions, *private* SPM of a task refers to the private SPM of the PE in which the task is running and with respect to an SPM spm_i, all tasks having private SPM spm_i are considered *local*. Similarly, *remote* SPM of a task refers to any SPM available in the MPSoC other than the *private* SPM of the task.

Computation of variable lifetime An interval $[lo(v), hi(v)]$ represents the lifetime of a variable v. $lo(v)$ indicates the *earliest* possible time v could possibly be accessed and $hi(v)$ indicates the *latest* possible time for an access to v. These intervals are computed initially and everytime after an SPM allocation decision is finalized. Under allocation A, $lo(v)$ and $hi(v)$ are computed as follows (assuming v is accessed in task t_i):

$$lo(v) = eStart(t_i, A) + \min_{r_i \in fref(v)} bcet(t_i, r_i, A) \tag{10.5}$$

$$\begin{aligned} hi(v) = \quad & lStart(t_i, A) + \max_{r_i \in lref(v)} wcet(t_i, r_i, A) \\ & + \sum_{t_j \in hp(t_i)} wcet(t_j, A) + |hp(t_i)| \times \mathcal{J} \times S_l \end{aligned} \tag{10.6}$$

Recall that $eStart(t_i, A)$ and $lStart(t_i, A)$ are the earliest and latest start times of task t_i under SPM allocation A. $fref(v)$ and $lref(v)$ represent the set of *first* and *last* references (in topological order) to variable v in task t_i respectively. $bcet(t_i, r_i, A)$ and $wcet(t_i, r_i, A)$ represent the *best case* and *worst case* execution time spent from the beginning of task t_i to reference r_i, under allocation A, respectively. To compute the *latest* reference time of variable v, we need to consider the preemption cost. Recall that $hp(t_i)$ represents the set of all tasks which may preempt task t_i and $|hp(t_i)| \times \mathcal{J} \times S_l$ bounds any additional bus delay introduced due to preemption. Therefore, Equation 10.6 finds the *latest* possible time at which the variable v is accessed.

Interference graph We use the lifetime information of variables to construct an interference graph $G_I = (V_I, E_I)$. Nodes of this graph correspond to different variables. Recall that $\mathcal{D}(t_i)$ denotes the set of tasks which depend (directly or indirectly) on task t_i . There exists an edge between two nodes depicting variables u (accessed in task t_i) and v (accessed in task t_j) if the following condition $pred_{uv}$ holds:

$$pred_{uv} = \quad [lo(u),\ hi(u)] \cap [lo(v),\ hi(v)] \neq \phi$$
$$\wedge\, t_i \notin \mathcal{D}(t_j) \wedge t_j \notin \mathcal{D}(t_i) \wedge u \neq v \tag{10.7}$$

The condition $pred_{uv}$ represents the scenarios where two different variables u and v might be *live* at the *same* time and thus cannot share the same memory space. As shown in the preceding, two variables from two dependent tasks can never interfere. If u and v are accessed by the same task t_i, number of edges in G_I is reduced by checking whether u and v can be simultaneously live using classical liveness analysis.

SPM allocation using the interference graph Interference graph is used for sharing the available SPM space as much as possible. Each node of the interference graph is assigned a weight. Nodes having higher weight values are given preference for SPM allocations. We want to place data items which incur high memory latency (including bus delay) into the SPM so that external memory access is not needed. At the same time, we want to optimize the critical path of the application and consequently, we want to place data items which are accessed in the critical path, into the SPM. Therefore, we assign a weight ($gain_v$) to each vertex (v) in the interference graph as follows:

$$gain_v = \begin{cases} 0, & \text{if } v \text{ is not accessed in the critical path} \\ & \textbf{or } v \text{ is allocated in SPM} \\ C_v, & \text{otherwise.} \end{cases} \tag{10.8}$$

These weights are computed initially and everytime an SPM allocation decision is made. Before going into the formal description of the technique, we define the following notations that will be used for rest of the discussion:

- $area : V_I \rightarrow \mathbb{N},\ area(v_i)$ denotes the size of a variable represented by interference graph node v_i.

- $size : 2^{V_I} \rightarrow \mathbb{N}$, $size(S)$ denotes the size of largest variable in set S that resides in external memory. Note that, if S forms an *independent set* in the interference graph, $size(S)$ is the total space needed to allocate the entire set of variables S into the SPM.
- $ref_i \subseteq V_I$: Set of variables accessed in some task assigned to PE i.
- spm_i : Private SPM of PE i.
- SP : Set of all SPMs available in the MPSoC.
- $capacity : SP \rightarrow \mathbb{N}$, $capacity(spm_i)$ denotes the free space in spm_i.
- $location : V_I \rightarrow \{SP \cup \perp\}$, $location(v_i)$ denotes the location of a variable v_i. If v_i is in external memory, $location(v_i) = \perp$.
- $\wp : V_I \rightarrow 2^{V_I}$, $\wp(v_i)$ denotes the set of variables sharing the same SPM space with v_i due to their disjoint lifetimes.

Formal description of the overall technique is given in Algorithm 4. There are mainly two decisions associated with every iteration of Algorithm 4: first, finding a set of variables for allocating in SPM (*maxIndependentSet* function in Algorithm 4) and secondly, finding space in shared SPM to allocate this set of variables (*findSPMspace* function in Algorithm 4). Broadly, our technique exhibits a search algorithm with limited backtracking. A choice made by the algorithm can be either *final* or can be *backtracked* depending on whether the choice improves application performance. The search algorithm terminates when no new choice can be made.

We apply *graph coloring* to the interference graph, the resulting colors will give us groups of variables which are accessed at disjoint time interval. Graph coloring using the minimum number of colors is known to be NP-complete. Therefore, we employ Welsh-Powell algorithm [95], a heuristic method that assigns the first available color to a node without restricting the number of colors to be used. Algorithm 4 follows a reduced backtracking technique. Let us define *weight* of a particular color CL as the sum of weights ($gain_v$) of all vertices colored with CL. Each color in the interference graph represents an independent set and the independent set corresponding to the maximum weighted color contribute a bigger chunk to the application's overall WCRT. Therefore, in each iteration of the algorithm, we choose a color that has the maximum weight. If allocating an independent set $\mathcal{IS}$ into SPM reduces the WCRT of the application, we finalize the allocation of $\mathcal{IS}$ and the location of variables representing set $\mathcal{IS}$ is never changed

Algorithm 4 *MIS*: SPM allocation by exploiting variable lifetime

1: Perform initial WCRT analysis to get the WCRT and critical path;
2: Construct interference graph G_I and assign weight $gain_v$ to all its vertices;
3: $backlog := \phi$;
4: **repeat**
5: **repeat**
6: $\mathcal{IS}_{max} := maxIndependentSet(G_I)$;
7: $gain := \sum_{v \in \mathcal{IS}_{max}} gain_v$;
8: /* $gain$ is reset in two conditions: (a) all variables in critical path are already allocated in SPM, (b) $G_I = \phi$, and consequently $\mathcal{IS}_{max} = \phi$. The allocation is terminated at this point */
9: **if** $(gain = 0)$ **then**
10: Finalize SPM allocation;
11: return;
12: **end if**
13: $(\mathcal{IS}, occ, R_{spm}) := findSPMspace(\mathcal{IS}_{max})$;
14: /* If SPM space cannot be found for set $\mathcal{IS}_{max}$, some previously allocated space is reclaimed if available. Otherwise, largest variable in $\mathcal{IS}_{max}$ is removed from G_I to find a smaller independent set */
15: **if** $(R_{spm} = \phi)$ **then**
16: $(\mathcal{IS}_m, occ_m, spm_i) := \max_{(*,occ,*)} backlog$;
17: **if** $(occ_m > 0)$ **then**
18: $capacity(spm_i) := capacity(spm_i) + occ_m$;
19: recompute the critical path and the weights $gain_v$;
20: $backlog := backlog \setminus (\mathcal{IS}_m, occ_m, spm_i)$;
21: **else**
22: $V_{max} := \{v_i \in \mathcal{IS} \mid area(v_i) = size(\mathcal{IS}_{max})\}$;
23: $G_I := G_I \setminus V_{max}$;
24: **end if**
25: **end if**
26: **until** $(R_{spm} \neq \phi)$
27: $capacity(R_{spm}) := capacity(R_{spm}) - occ$;
28: recompute the critical path and the weights $gain_v$;
29: **if** (WCRT is reduced after allocating $\mathcal{IS}$ in R_{spm}) **then**
30: $backlog := \phi$;
31: recompute G_I;
32: **else**
33: /* remove the previously selected independent set from the interference graph and continue allocation with the remaining graph */
34: $backlog := backlog \cup \{(\mathcal{IS}, occ, R_{spm})\}$;
35: $G_I := G_I \setminus \mathcal{IS}$;
36: **end if**
37: **until** $(G_I = \phi)$

further. However, if allocating $\mathcal{IS}$ into SPM does not reduce the WCRT, we maintain it in a list *backlog* as long as enough SPM space is available for WCRT improvement. When we run out of space, we search through the *backlog* list to find a victim and reclaim the SPM space assigned

to it. The victim is chosen to be the one which occupies maximum amount of space among all other elements in *backlog* list (the term $\max_{(*,occ,*)} backlog$ in Algorithm 4 computes this victim). If the list *backlog* is empty and there is not enough SPM space to allocate an independent set, the largest variable from the chosen independent set is removed to find a smaller independent set that can be accommodated in free SPM space. Size of *backlog* list represents the maximum depth of backtracking. One could argue about the backtracking depth being nonzero (for zero backtracking depth, an independent set is never allocated to shared SPM unless it reduces the overall WCRT). However, we observe that more than one independent sets (say $\mathcal{IS}_1$ and $\mathcal{IS}_2$) are often able to reduce the WCRT if allocated together into the SPM, whereas, WCRT might not reduce if either $\mathcal{IS}_1$ or $\mathcal{IS}_2$ is allocated to SPM but not both. Therefore, even if the WCRT is not reduced after an allocation decision, we expect that WCRT will reduce in future iterations and we only discard such decision when there is not enough space for a new allocation (recall that allocation decisions that did not lead to WCRT improvement, are maintained in a separate list *backlog*). The above-mentioned situation is encountered very often when the cardinality of an independent set is very small or the expected *gain* from the corresponding allocation decision is low. Consequently, WCRT may improve only by allocating more than one independent sets together. Finally, the interference graph G_I is recomputed only if the WCRT is reduced. This is to ensure that the set of edges in G_I monotonically decreases — a crucial property that maintains the correctness of our algorithm (Theorem 10.5.2).

We use a heuristic as described in Algorithm 5 to find SPM space for a given independent set $\mathcal{IS}_{max}$. Note that we only need to find an SPM to allocate the independent set $\mathcal{IS}_{max} \setminus V_{spm}$, where $V_{spm} (\subseteq \mathcal{IS}_{max})$ is a set of variables already allocated in the SPM space. Choosing an SPM for allocating a group of non-interfering variables has a space vs quality trade-off. Since, a group of variables are sharing the space, it will create opportunities for more variables to be accommodated in SPM. On the other hand, as the interference graph is a globally unified graph, a group may consist of variables that are accessed in different processors. Therefore, if the group is allocated the same space, some variables in the group might be accessed *remotely* and thereby limit the WCRT improvement. Since a very limited amount of SPM is normally available in a processor, our primary focus is to utilize the available space with maximum possible sharing. Therefore, in the first step of our heuristic, we check whether the set of variables $\mathcal{IS}_{max}$ can share SPM space with some variables already allocated in SPM. However, if our first step is unsuccessful, we try to improve the WCRT by minimizing the latency incurred by the *costliest*

Algorithm 5 *findSPMspace:* Finding SPM space for a set of variables $\mathcal{IS}_{max}$ having disjoint lifetimes. Total number of processors is $\mathcal{J}$.

1: /* If some variable $\in \mathcal{IS}_{max}$ is already allocated in SPM, it is checked whether the space can further be shared with the current set of variables $\mathcal{IS}_{max}$ */
2: $V_{spm} := \{v_i \in \mathcal{IS}_{max} \mid location(v_i) \in SP\}$;
3: $\mathcal{IS} := \mathcal{IS}_{max} \setminus V_{spm}$;
4: /* Required space in SPM for independent set $\mathcal{IS}_{max}$ */
5: $occ := size(\mathcal{IS}_{max})$;
6: **if** $(\exists v_i \in V_{spm}.\ occ \leq area(v_i) \wedge \bigwedge_{x,y \in \mathcal{IS} \cup \wp(v_i)} \neg pred_{xy})$ **then**
7: **return** $(\mathcal{IS}, 0, location(v_i))$;
8: **end if**
9: /* Try to minimize the latency incurred by the costliest subgroup in $\mathcal{IS}_{max}$ */
10: $cg(i) := \sum_{v \in \mathcal{IS}_{max} \cap ref_i} gain_v\ ,\ \forall i \in [1, \mathcal{J}]$;
11: **if** $(\exists i \in [1, \mathcal{J}].\ cg(i) = \max_{k \in [1, \mathcal{J}]} cg(k) \wedge capacity(spm_i) \geq occ)$ **then**
12: **return** $(\mathcal{IS}, occ, spm_i)$;
13: **end if**
14: /* Find a scratchpad having maximum remaining space and has least interference from locally executing critical tasks */
15: $intf := \{u \mid u \in V_I \wedge \exists v \in \mathcal{IS}.\ pred_{uv}\}$;
16: $hr(spm_i) := \frac{capacity(spm_i) - occ}{\sum_{v \in intf \cap ref_i} gain_v}\ ,\ \forall i \in [1, \mathcal{J}]$;
17: **if** $(\exists i \in [1, \mathcal{J}].\ hr(spm_i) \geq 0 \wedge hr(spm_i) = \max_{k \in [1, \mathcal{J}]} hr(spm_k))$ **then**
18: **return** $(\mathcal{IS}, occ, spm_i)$;
19: **end if**
20: **return** $(\mathcal{IS}, 0, \phi)$;

subgroup in $\mathcal{IS}_{max}$. A costliest subgroup is a set of variables in $\mathcal{IS}_{max}$ that are accessed in the same processor and have maximum *cumulative weight* (sum of the elements' weight $gain_v$). Consequently, if sufficient space is available, we allocate $\mathcal{IS}_{max} \setminus V_{spm}$ in the private SPM of the processor accessing this costliest subgroup. In our final step, we choose an SPM that has the maximum remaining space and has minimum interference with $\mathcal{IS}_{max} \setminus V_{spm}$ from locally executing critical tasks. We try to minimize the possibility of high interference in the private SPMs of critical tasks at this final stage.

Following three theorems highlight certain crucial properties of our allocation technique.

Theorem 10.5.1. *Set of edges in the interference graph monotonically decreases over different iterations of Algorithm 4.*

Proof. We prove this by contradiction. In Algorithm 4, we recompute G_I if and only if WCRT is reduced. Let us assume a specific recomputation of G_I as $G_I^{m+1} = (V_I^{m+1}, E_I^{m+1})$ and assume that the set of variables allocated to SPM is A_{m+1}. Further assume, the immediate

last recomputation of G_I was $G_I^m = (V_I^m, E_I^m)$ and had a set of SPM-allocated variables A_m. Clearly, $A_m \subseteq A_{m+1}$ and $V_I \setminus A_{m+1} \subseteq V_I \setminus A_m$ where V_I is the set of all variables. More over, locations of the set of variables A_m are never changed after computing G_I^m. By contradiction, assume $E_I^m \subset E_I^{m+1}$. When computing WCRT with allocation A_m (A_{m+1}), location of the set of variables $V_I \setminus A_m$ ($V_I \setminus A_{m+1}$) is taken as off-chip memory to exploit the worst-case scenario. On the other hand, BCET computation under allocation A_m (A_{m+1}) takes the location of the set of variables $V_I \setminus A_m$ ($V_I \setminus A_{m+1}$) as private SPM to exploit the best-case situation. Close inspection of Equation 10.7 reveals that the property $E_I^m \subset E_I^{m+1}$ can only be satisfied in following two conditions: first, WCRT of some task (or task fragment) is comparatively higher with allocation A_{m+1} than with allocation A_m. It is not possible as $A_m \subseteq A_{m+1}$ and on-chip SPMs have lower latencies than off-chip memory. Secondly, BCET of some task (or task fragment) is more with allocation A_m than with allocation A_{m+1}. By a similar reasoning we argue that it is also not possible as $V_I \setminus A_{m+1} \subseteq V_I \setminus A_m$ and private SPM has the lowest latency. □

Theorem 10.5.2. *Set of variables sharing the same space in SPM can never have interfering lifetimes across different iterations of SPM allocation in Algorithm 4.*

Proof. Two variables v_i and v_j could be allocated at the same space in shared SPM only if the edge $(v_i, v_j) \notin E_I$. However, according to Theorem 10.5.1, set of edges in G_I monotonically decreases. Therefore, the property $(v_i, v_j) \notin E_I$ must be satisfied in all future iterations of Algorithm 4 (i.e., after the iteration where v_i and v_j had been alloted the same SPM space). Consequently, set of variables occupying the same space can never have interfering lifetimes. □

Time complexity We propose the following theorem to analyze the complexity of our iterative allocation framework:

Theorem 10.5.3. *Let us assume $|V_I|$ is the total number of variables in the interference graph. Total number of iterations in our framework (bound of the outer loop in Algorithm 4) cannot exceed $\frac{|V_I|(|V_I|+1)}{2}$.*

Proof. Let us assume that after a specific recomputation of G_I, X is the number of variables residing in off-chip memory and $T(X)$ is the number of remaining iterations in Algorithm 4. In the worst case scenario, $T(X)$ follows the recurrence $T(X) = T(X-1) + X$. In

the worst case, interference graph could be a complete graph in every iteration, making the size of selected independent set by $maxIndependentSet$ exactly 1. Consequently, at most X iterations might be required to finalize an SPM allocation decision (because all previous $X-1$ choices may not lead to WCRT reduction and subsequently put into the $backlog$ list). Assume that the variable v_X is chosen for SPM allocation at X-th iteration. Note that, if WCRT is not improved after allocating v_X, Algorithm 4 will be terminated. Similarly, if WCRT improves after successfully allocating all X variables in SPM, Algorithm 4 also terminates as there are nothing more to allocate in SPM. Therefore, to visualize the worst case situation, we assume that $backlog$ list is emptied out at X-th iteration to accommodate v_X in SPM and allocation of v_X improves the WCRT. Since allocation of v_X leads to $X-1$ variables in off-chip memory, it will require $T(X-1)$ iterations more for Algorithm 4 to terminate. Solving the recurrence we get $T(X) = \frac{X(X+1)}{2}$. Since there are a total of $|V_I|$ nodes in the interference graph, maximum number of iterations in Algorithm 4 is bounded by $\frac{|V_I|(|V_I|+1)}{2}$. □

In practice, though, above theoretical bound is not reached. It is mostly because of the two reasons: first, the interference graph is hardly a complete graph in *any* iteration and secondly, search depth to finalize an allocation decision is much lower than the number of variables residing in off-chip memory. We shall see in the experimental section that our framework converges quickly.

10.6 Experimental evaluation

Benchmarks We have used two real-life embedded applications to evaluate our scratchpad allocation schemes. Our first case study corresponds to a large fragment of DEBIE-I DPU Software [82], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. We model this fragment as a task graph, shown in Figure 10.5. The number beside each task in Figure 10.5 shows the assignment of tasks to different PEs. Code size of the tasks varies from 448 bytes to 23288 bytes (average code size 8825 bytes) whereas the data size varies from 18 bytes to 66972 bytes (average data size 55448 bytes).

Our second case study is the Unmanned Aerial Vehicle (UAV) control application from *papabench* [96], a derivation from the real-time embedded UAV control software Paparazzi. The controller consists of two main functional units, *fly_by_wire* and *autopilot*, which are interconnected by SPI serial link. *fly_by_wire* unit is responsible for managing radio-command orders

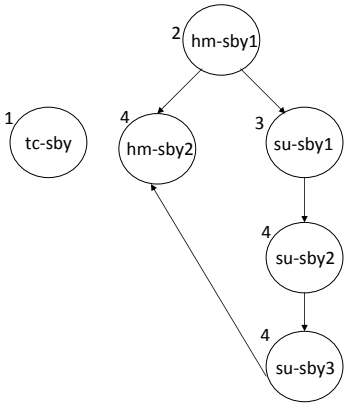


Figure 10.5: Task graph extracted from DEBIE-DPU

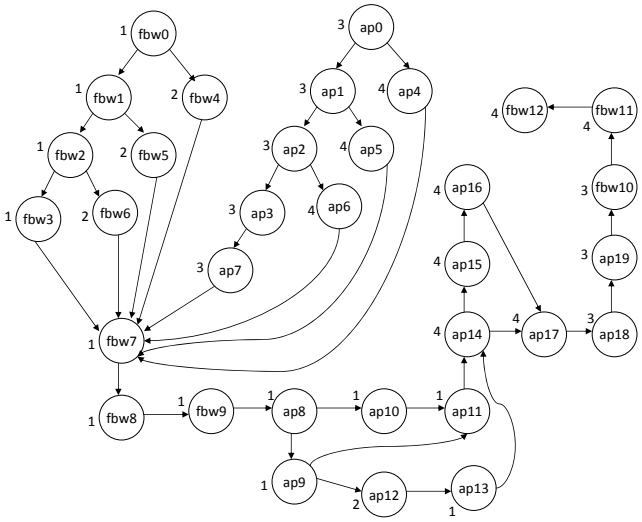


Figure 10.6: Task graph of *papabench*

and servo-commands, while *autopilot* runs the navigation and stabilization tasks of the aircraft. One scenario in the manual mode is modeled as a task graph and is shown in Figure 10.6. The number beside each task in Figure 10.6 shows the assignment of tasks to different PEs. Code size of the tasks varies from 96 bytes to 6468 bytes (average code size 1903 bytes) whereas the data size varies from 130 bytes to 1878 bytes (average data size 1105 bytes).

Experimental setup We have implemented our allocation algorithm inside a cycle accurate WCRT analyzer. Our full experimental setup is shown in Figure 10.7. We shall use the terminologies shown in Figure 10.7 for rest of the discussion in this section. Let us assume *WC* represents the scenario where all variables are accessed from external memory. Similarly, *BC* represents the scenario where all variables are accessed from private SPM. Therefore, *WC* and *BC* provide upper and lower bound of optimized WCRT value respectively. To check the improvement by our SPM allocator (result shown by "*MIS*" in Figure 10.7), we measure the ratio

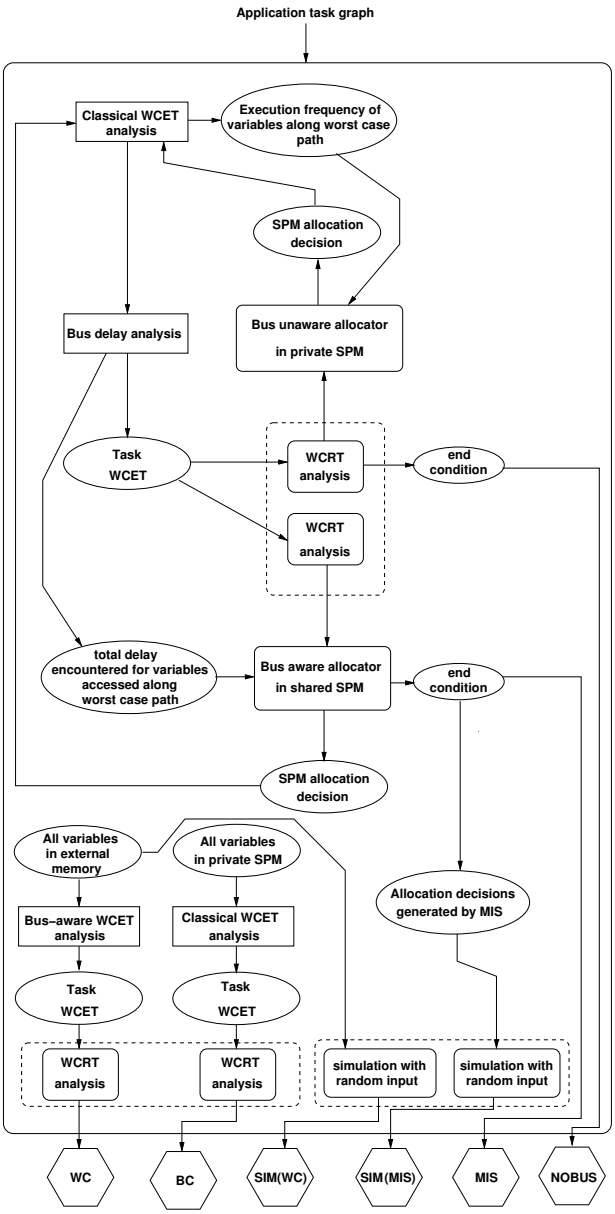


Figure 10.7: Experimental setup

$\frac{WC}{MIS} - 1$. Clearly, *BC* is a measurement of best possible scenario when all variables are accessed from private SPM and $\frac{WC}{BC} - 1$ bounds the best possible improvement. We compare our improved WCRT with a *bus-unaware* allocator (result shown by "*NOBUS*" in Figure 10.7) that optimizes the content selection in individual private SPMs (similar to the SPM allocator described in [63]). Improvement from *NOBUS* is similarly measured as $\frac{WC}{NOBUS} - 1$. We also check the effect of our Algorithm on average case response time (ACRT) by running the application using random inputs with and without our final SPM allocation decision (results shown by "*SIM(MIS)*" and "*SIM(WC)*" in Figure 10.7 respectively). Both "*SIM(WC)*" and "*SIM(MIS)*" are obtained using a cycle accurate simulator. ACRT improvement using our technique is measured

	Size of interference graph		Analysis statistics		
Benchmark	Nodes	Edges	Iterations	Time	WC (cycles)
debie	283	27688	99	88 secs	3773×10^6
papabench	506	16872	210	119 secs	515×10^3

Table 10.1: Problem size, analysis time and WCRT

as $\frac{SIM(WC)}{SIM(MIS)} - 1$.

We assume a single in-order pipeline for each processor. Each processor can access its private SPM in a single cycle. Our experimental results (*i.e.* WCRT of the application) mainly depend on four different micro-architectural parameters whose default values are configured as follows: i) total size of shared scratchpad space (relative to total data size in the application): 10%, ii) remote SPM latency: 4 cycles, iii) off-chip memory latency: 30 cycles, iv) round-robin TDMA bus slot length: 50 cycles and v) number of PEs: 4. We have carried out experiments with two processors and with four processors. For all experiments with two PEs, we combine the tasks running in PE 2 and PE 3 to run in one PE and combine rest of the tasks to run in another PE. We perform all our experiments in a 3 GHz Pentium IV machine having 1 GB of RAM and running Ubuntu 8.10 as the operating system. Table 10.1 gives an idea about the problem size and time taken by our iterative SPM allocator in default configuration. The time shown in Table 10.1 features the total time, including the time taken by repeated computations of bus-aware WCET and SPM allocation decisions by Algorithm 4. In general, none of our reported experiments takes more than 2 minutes to complete.

Sensitivity of WCRT reduction with respect to SPM size Figures 10.8(a)-10.8(b) demonstrate WCRT improvement for different SPM size. SPM size is chosen in a way such that sufficient amount of interferences take place among all data items (to accommodate them in shared SPM space). Above figures clearly demonstrate that we can obtain significant WCRT reduction by using our SPM allocator. For *debie*, an upper bound on WCRT improvement or the measured ratio $\frac{WC}{BC} - 1 \times 100\%$ is 1500% (700%) using 4 PEs (2 PEs). Similarly, for *papabench*, the upper bound on WCRT improvement is 710% (410%) using 4 PEs (2 PEs). WCRT is consistently improved with bigger SPM size, which is expected as the interferences among data items reduce with bigger SPM size. Interferences among data items also reduce when more processors are used. Since *debie* has much smaller number of tasks compared to *papabench*, the reduction in interferences for *papabench* is much higher compared to *debie* when more processors are used. We observe the situation in our result – for *debie*, WCRT improvement hardly gets affected with

more processors, whereas for *papabench*, WCRT is improved upto 100% when 4 processors are used instead of 2. Finally, our SPM allocator can improve the WCRT considerably compared to a bus-unaware allocator — with a maximum improvement being more than 60%.

Sensitivity of WCRT reduction with respect to bus slot length We have also measured the sensitivity of our allocator with bus slot length. This measurement is shown in Figures 10.8(c) -10.8(d). Average improvement from our SPM allocator is 52% (46%) for *debie* (*papabench*) when compared with a bus-unaware SPM allocator over a bus slot length range of 40-80 cycles.

Summary of other results To test the robustness of our approach, we have measured its sensitivity with different remote SPM latencies. Figure 10.8(e) demonstrates this result both for *papabench* and *debie*. In contrast to the bus unaware allocator, WCRT improvement from our SPM allocator decreases with increased remote SPM latency, as fetching memory blocks from remote SPM now takes more time. Nevertheless, the rate of decrement is quite low (maximum 5%) and the average improvement over bus unaware allocator remains at 55% over a range of 4-12 cycles remote SPM latency. We have also checked the WCRT improvement by varying off-chip memory latency. When checked with different off-chip memory latencies over a range of 10-50 cycles, the percentage reduction in WCRT remains similar (to Fig. 10.8). Finally, we have also measured the effect of our WCRT oriented optimization on average case response time (ACRT). Unfortunately, the inputs to *debie* are not available in public domain, which prevents us from running simulation and producing ACRT in *debie*. Therefore, we present the result of ACRT improvement for *papabench* (in Figure 10.8(f)). As our SPM allocator aims to optimize WCRT and the critical path, we observe that reduction in ACRT is not much compared to the same in WCRT. As evidenced by Figure 10.8(f), ACRT is reduced by 130% on average over a varying range of scratchpad size.

10.7 Extensions and Future Work

Applications using shared variables Our current implementation does not handle shared variables among different tasks. More precisely, our allocation framework only considers the set of variables which are accessed by *exactly one* task. However, our allocation method can be modified to deal with shared variables as follows: first, there will be *at most one copy* of each shared variable in the SPM space to maintain *coherency*. Secondly, a shared variable may be

accessed by *critical* as well as *non-critical* tasks. Therefore, when we compute the metric $gain_v$ (refer to Equation 10.8) for a shared variable v, $gain_v$ is set to be the sum of C_v values only in the critical tasks (i.e., references to shared variable v in all non-critical tasks are ignored). Thirdly, lifetime of a shared variable must take into account all the tasks (in application) in which the shared variable might possibly be accessed. In future, we plan to extend our work to include shared variables.

Other multi-processor architectures Our underlying architecture contains a crossbar to access fast on-chip memories. However, some of the architectures [97] use a fast on-chip bus for accessing a remote SPM. Since the on-chip buses operate on high bandwidth, remote SPM latency is still bounded by a small constant. Consequently, our SPM allocation framework can be applied without modification.

ACRT optimization Interference graph in our SPM allocation framework is used for finding a group of variables having disjoint lifetimes. Therefore, interference graph can also be used for other kind of optimization which allows SPM space sharing among different variables. Only driving factor for WCRT oriented optimization is the assigned $gain_v$ metric for each variable v. For ACRT optimization, a *trace* can be collected using a simulator, which will include the external memory access profile along the most frequently accessed path π. $gain_v$ will represent the total latency incurred (including the bus delay) to access variable v along π. Only non-trivial task is to efficiently recompute $gain_v$ after an allocation decision. In future, we plan to check the efficacy and scalability of our allocation framework for ACRT guided optimization.

10.8 Chapter summary

In this Chapter, we have discussed how our analysis framework can be used to achieve execution time predictability through compiler optimization. Specifically, we have presented a scratchpad allocation framework for multi-processor system-on-chip (MPSoC) platforms. The prime novelty in our work is to incorporate the bus schedule into the multi-processor scratchpad allocation scheme. Our allocation framework exploits the shared scratchpad space available in MPSoCs, and considers variable lifetimes to efficiently utilize the available shared scratchpad space. As evidenced by our experiments, our scratchpad allocation scheme is able to significantly reduce the WCRT of real-life embedded applications. Our results are also considerably better when

compared with an existing SPM allocation framework. Our allocation method is efficient and thus the scalability of our framework is evident.

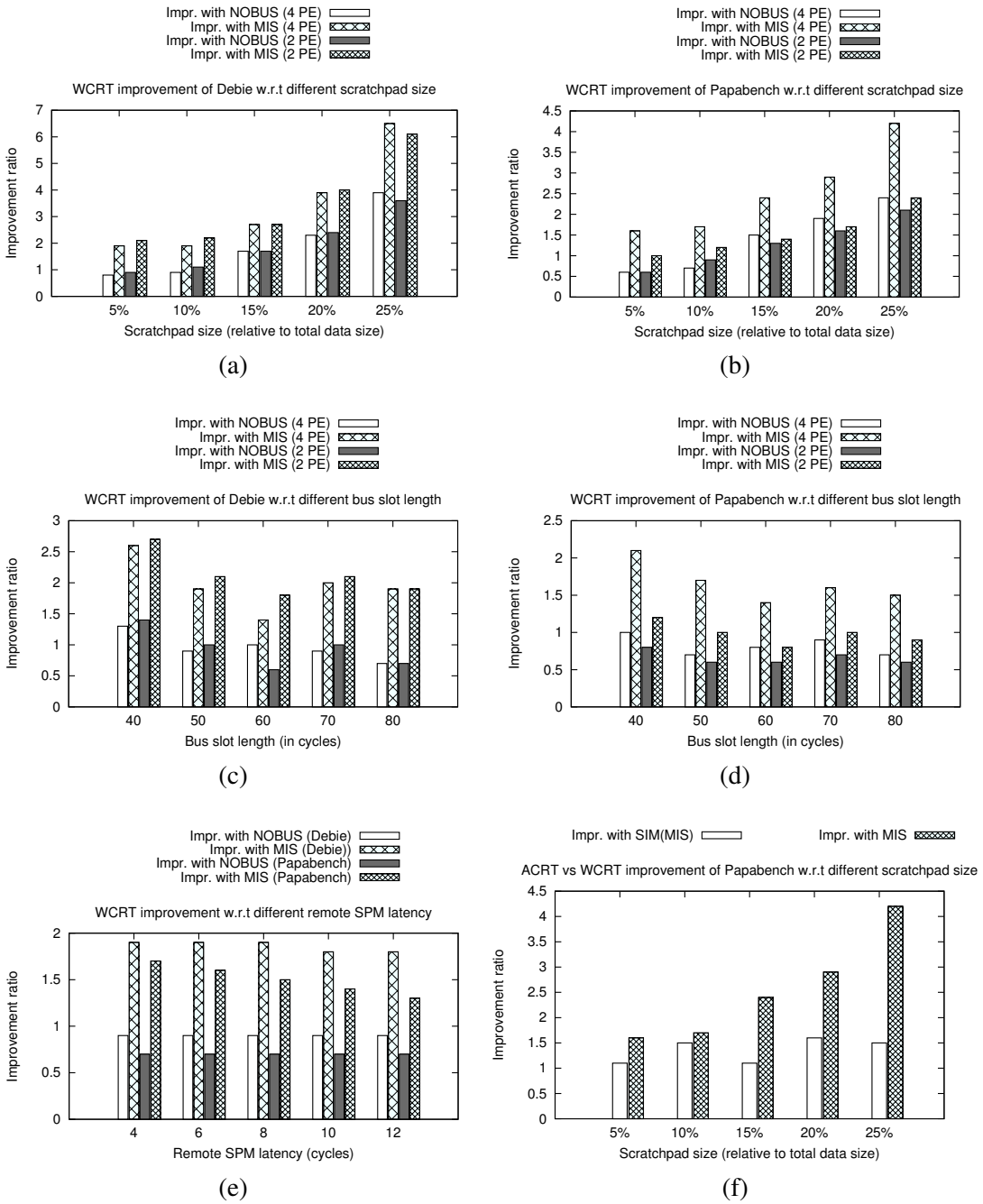


Figure 10.8: Experimental evaluation of our allocation framework

Chapter 11

Discussion and Future Work

In this chapter, we shall conclude this dissertation by briefly summarizing the contribution and pointing to a number of possible future directions.

Contribution

In this dissertation, we have primarily focused on the problem of execution time predictability in multi-core platform. Our contribution in this direction is two-fold: first, we have modeled and developed an analysis framework for statically predicting the *worst case execution time* (WCET) of a program running on multi-core. Our analysis framework takes into account the modeling of key shared resources in multi-core (*e.g.* shared cache and shared bus) and it is also capable of analyzing the complex timing interactions among the shared resources and basic micro-architectural features. We have performed a detailed evaluation of our analysis framework to show its usefulness for predicting the WCET of a program. We have also shown how such an analysis framework can be used to find the different sources of WCET overestimation in multi-core platform. The second primary contribution of this dissertation is to show the applicability of this analysis framework in a popular compiler optimization. We have shown that the analysis result can be used to perform scratchpad allocation in customized MPSoCs, which in turn reduces the timing unpredictability arising due to shared bus traffic.

Future work

Extension of basic multi-core WCET analysis Even though we have provided a basic framework for WCET analysis in multi-cores, it is worthwhile to mention a few important extensions to this basic framework. One such extension could be the modeling of I/O peripherals. In this

dissertation, we have only considered the operations performed by a processor. In the presence of I/O peripherals, additional interferences might exist, such as in shared caches and in shared buses. Such interferences are caused by concurrent requests from processors and I/O peripherals [44]. Therefore, modeling such interferences will be an important contribution towards a more accurate estimation of WCET. Other important extensions could be the integration of advanced micro-architectural features, such as branch target buffers (*e.g.* using [98]), load-store units [99] into our basic WCET analysis framework. Our current WCET analysis tool for multi-core does not include the modeling of data caches. In future, we plan to integrate the modeling of data caches (*e.g.* using [27]) into our multi-core, WCET analysis framework. Finally, for multi-threaded programs, synchronization constructs are generally used to protect shared variable accesses. In this dissertation, we have only modeled the cache coherence misses. However, in the presence of synchronization constructs, a thread may spend time in acquiring the respective synchronization locks - leading to additional delay in execution. The delay introduced due to the synchronization constructs have not been modeled in our framework. Modeling such delay will also be an important contribution towards the WCET analysis of multi-threaded embedded software.

Customized hardware for real-time embedded systems With the ever increasing demand of embedded systems, multi-core processors are quickly being adopted in the embedded computing world. Our dissertation has looked into the challenges and their possible solutions in moving towards multi-processing for hard real-time systems. We hope that our work will inspire future research in adopting multi-processing for hard real-time systems. We believe that the ideas developed in this dissertation can be used to build customized hardware for running hard real-time applications. Such a customized hardware should target to design the part of hardware for time-predictable execution with acceptable loss of performance. The time predictability at the different stages of the design can be computed by a similar analysis framework proposed in this dissertation. Whereas, the research community has already looked into building time predictable hardware [43; 44], previous approaches had mostly ignored the benefit of any WCET analysis framework targeted towards multi-core and have tried to reuse the WCET analysis in single core. Therefore, we believe a combined approach of multi-core WCET analysis and designing time-predictable hardware will lead to a notable contribution towards adopting multi-processing for hard real-time computing.

Energy estimation for real-time computing Whereas in our dissertation we have focused on performance analysis, there exists another critical aspect related to hard real-time computing — namely the energy consumption. It is well known that the complexity of a processor has grown several magnitudes greater than the battery technology has evolved. When today's mobile devices are using multi-processing in fast growing pace, it is still of great concern to run any critical application in such mobile devices. The battery life poses a threat to run such critical applications, as it is clearly undesirable if the mobile device gets switched off while running a critical application. Therefore, the prediction of energy consumption is also of prime importance for such applications. In past years, worst-case energy estimation has been proposed in [100] for single core processors. The solution proposed in [100] has used the progress in single core WCET analysis research to estimate energy for real-time applications. Therefore, we hope that the techniques developed in this dissertation will inspire the future research of adopting advanced mobile devices for real-time applications.

Bibliography

[1] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven cache-based procedure positioning optimizations. In *ECRTS*, 2008.

[2] WCET benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[4] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, pages 358–363, 2006.

[5] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, 2006.

[6] C. A. Healy, M. Sjödin, V. Rustagi, D. B. Whalley, and R. v. Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3), 2000.

[7] D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, 2009.

[8] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3), 1999.

[9] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[11] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.

[12] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[13] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *SAS*, 2002.

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

[15] Y. Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[16] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS, 1999.

[17] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3), 2006.

[18] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1), 2005.

[19] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1), 1999.

[20] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES*, 2001.

[21] Ilog, Inc. Solver CPLEX, 2003. `http://www.ilog.fr/products/cplex/`.

[22] L. Ju et al. Performance debugging of Esterel specification. In *CODES+ISSS*, 2008.

[23] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. `http://www.comp.nus.edu.sg/~rpembed/chronos`.

[24] aiT AbsInt. `http://www.absint.com/ait`.

[25] R.T. White, C.A. Healy, D.B. Whalley, F. Mueller, and M.G. Harmon. Timing analysis for data caches and set-associative caches. In *RTAS*, 1997.

[26] R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.

[27] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.

[28] C.G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6), 1998.

[29] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.

[30] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.

[31] Y. Tan and V.J. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES*, 2004.

[32] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*, 2004.

[33] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, 2009.

[34] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *LCTES*, 2010.

[35] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, 2008.

[36] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR*, 1999.

[37] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

[38] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7), 2009.

[39] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*, pages 103–110, 2008.

[40] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.

[41] T. Kelter et al. Bus aware multicore WCET analysis through TDMA offset bounds. In *ECRTS*, 2011.

[42] M. Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.

[43] M. Paolieri et al. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, 2009.

[44] R. Pellizzoni et al. A predictable execution model for cots-based embedded systems. In *RTAS*, 2011.

[45] I. Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *ECRTS*, 2006.

[46] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, 1989.

[47] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Trans. Computers*, 42(8), 1993.

[48] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *RTSS*, 2003.

[49] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS*, 2003.

[50] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *RTSS*, 2004.

[51] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *IEEE Real-Time Systems Symposium*, 2009.

[52] 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 *CODES*, 2002.

[53] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, 2003.

[54] P.R. Panda, N.D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, 2000.

[55] P. R. Panda, A. Nicolau, and N. Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. 1998.

[56] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1), 2002.

[57] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.

[58] J.-F. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *ECRTS*, 2007.

[59] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, 2002.

[60] H. Falk and J.C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *DAC*, 2009.

[61] M.T. Kandemir, J. Ramanujam, and A.N. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, 2002.

[62] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *CASES*, 2006.

[63] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. *ACM Trans. Program. Lang. Syst.*, 32(4), 2010.

[64] M. Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3), 2007.

[65] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *IEEE Real-Time Systems Symposium*, 2008.

[66] F. Mueller. Timing predictions for multi-level caches. In *LCTES*, 1997.

[67] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *WCET*, 2009.

[68] M. Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.

[69] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, 2011.

[70] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[71] E. Clarke et al. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19, 2001.

[72] G. Balakrishnan et al. Model checking x86 executables with codesurfer/x86 and wpds++. In *CAV*, 2005.

[73] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[74] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[75] KLEE. The KLEE Symbolic Virtual Machine. `http://klee.llvm.org`.

[76] LLVM. The LLVM compiler infrastructure. `http://llvm.org`.

[77] STP. The STP Constraint Solver. `http://sites.google.com/site/stpfastprover`.

[78] ARM. ARM Cortex-A9 MPCore processor. `http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf`.

[79] G. Varghese et al. Penryn: 45-nm next generation Intel core-2 processor. In *IEEE Asian Solid-State Circuits Conf.*, 2007.

[80] S. Tam Rusu et al. A 65-nm Dual-Core Multithreaded Xeon processor with 16-MB L3 Cache. *IEEE Journal Of Solid State Circuits*, (1), 2007.

[81] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.

[82] European Space Agency. DEBIE – First standard space debris monitoring instrument, 2008. Available at: `http://gate.etamax.de/edid/publicaccess/debie1.php`.

[83] Intel. Intel Core-2 Duo Processor. `http://www.intel.com/products/processor/core2duo/index.htm`.

[84] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2), 2007.

[85] D. Grund and J. Reineke. Abstract interpretation of FIFO replacement. In *Static Analysis Symposium*, 2009.

[86] D. Grund and J. Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Euromicro Conference on Real-Time Systems*, 2010.

[87] D. Hardy and I. Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture - Embedded Systems Design*, 57(7), 2011.

[88] C. Berg. PLRU cache domino effects. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[89] D. Grund and J. Reineke. Toward precise PLRU cache analysis. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2010.

[90] M. M. Zahran, K. Albayraktaroglu, and M. Franklin. Non-inclusion property in multi-level caches revisited. *I. J. Comput. Appl.*, 14(2), 2007.

[91] Chronos for multi-cores: a WCET analysis tool for multi-cores. `http://www.comp.nus.edu.sg/~rpembed/chronos-multi-core.html`.

[92] S. Chattopadhyay et al. A unified WCET analysis framework for multi-core platforms. In *RTAS*, 2012.

[93] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *WCET Workshop*, 2006.

[94] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6), 1990.

[95] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 1967.

[96] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *WCET Workshop*, 2006.

[97] M. Gschwind. The Cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3), 2007.

[98] D. Grund, J. Reineke, and G. Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture*, 57(6), 2011.

[99] M. A. Maksoud and J. Reineke. An empirical evaluation of the influence of the load-store unit on WCET analysis. In *WCET*, 2012.

[100] R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *IEEE Real Time Technology and Applications Symposium*, 2006.